Public Key Cryptography 2

RSA

Reference: Rivest, Shamir, Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, CACM, Vol. 21, No. 2, pp. 120–126, February 1978.

RSA is a public key cryptosystem based on number theory.

The security of RSA is based on the **difficulty** of **factoring** a number to its prime factors, while its efficiency is based on the ease of multiplying prime numbers and checking whether given numbers are primes.

RSA — the Key Generation

User A chooses his keys by:

1. Randomly chooses two large prime numbers p and q of size 512 bits (154 decimal digits) at least.
2. Computes $n = pq$ (a 1024-bit number).
3. Randomly chooses an odd number e in the range $1 < e < \varphi(n)$ which is coprime to $\varphi(n)$ (i.e., $e \in Z^*_{\varphi(n)}$).
4. Computes $e \equiv d^{-1} \pmod{\varphi(n)}$ by Euclid's algorithm. Thus, $de \equiv 1 \pmod{\varphi(n)}$.
5. Publishes e, n as the public key, and keeps d secret as the secret key. (There is no need to keep p, q and $\varphi(n)$).

We denote the public key of user A by e_A, n_A and the secret key by d_A.

RSA — the Key Generation (cont.)

Notes:

1. It is easy to find random primes: One of every $\ln n$ numbers around n is a prime. Given a number, it is easy to check whether it is a prime (by a probabilistic algorithm). Therefore, to choose a random prime, random numbers are chosen and are checked whether they are primes. On average about $\ln n$ number are chosen and checked till a prime is found.

2. Choosing e: In $Z^*_{\varphi(n)}$ there are $\varphi(\varphi(n))$ invertible numbers modulo $\varphi(n)$, and we do not choose even e's (which are not invertible). Thus, about $\frac{\varphi(n)}{2\varphi(\varphi(n))}$ random e's should be chosen till an invertible e is found — in most cases the first or second chosen e is selected.

RSA — Encryption/Decryption

The encryption algorithm E:
Everybody can encrypt messages m $(0 \leq m < n_A)$ to user A by

$$c = E_A(m) = m^{e_A} \bmod n_A.$$

The ciphertext c $(0 \leq c < n_A)$ can be sent to A, and only A can decrypt.
The decryption algorithm D:
Only A knows his secret key d_A and can decrypt:

$$m = D_A(c) = c^{d_A} \bmod n_A.$$

RSA — Correctness

Theorem:

$$\forall m \in Z_n \qquad D(E(m)) = m.$$

Proof: We should prove that

$$\forall m \in Z_n \qquad (m^e)^d \equiv m \pmod{n}.$$

It suffices to prove the congruence twice: modulo p and modulo q. Without loss of generality we prove modulo p.

1. if $p|m$: $m^{ed} \equiv 0^{ed} \equiv 0 \equiv m \pmod{p}$.
2. if $p \not| m$:

$$m^{ed} \equiv m^{1+k(p-1)(q-1)} \equiv m(m^{p-1})^{(q-1)k} \equiv m \pmod{p}.$$

QED

RSA — Efficiency

Key generation:

1. It is easy to test random numbers for primality (using probabilistic algorithms; a less efficient deterministic algorithm also exists).

2. It is easy to invert numbers in Z_n using Euclid's algorithm.

RSA — Efficiency (cont.)

Encryption and decryption:

1. Efficient modular exponentiation to the exponent e requires about $1.5 \log e$ multiplications, where each multiplication is modular as well (all intermediate results are not larger than the modulus).

2. To increase efficiency of encryption, relatively small e's can be used.

3. To increase efficiency of decryption (by a factor of about 4), it is possible to decrypt modulo p and q separately, and combine the results using the Chinese remainder theorem.

In practice, key generation takes up to a few seconds, and encryption/decryption can be performed thousands of times every second on modern computers.

RSA — Strength

Note that if $p|m$ (or $q|m$) and $m \neq 0$ then $\gcd(m,n) = p$ (or q). In such a case, the user can compute p, q and the secret key d.
We now show that the probability of it is low: The number of numbers coprime to n in Z_n is $\varphi(n)$. Thus, the number of non-coprime numbers is

$$n - \varphi(n) = pq - (p-1)(q-1) = pq - pq + p + q - 1 = p + q - 1,$$

and the probability to have a non-coprime number is

$$\frac{n - \varphi(n)}{n} = \frac{p + q - 1}{n} \approx \frac{2^{512} + 2^{512}}{2^{1024}} = 2^{-511}$$

(or less if $|n| > 1024$). Therefore, it is very improbable that m and n are not coprime.
We assume that **factoring** is difficult. Otherwise, from the factors p and q of n it is easy to compute $\varphi(n)$ and d.

RSA — Strength (cont.)

Theorem: Given $\varphi(n)$ it is easy to compute p and q.
Proof: Given $\varphi(n)$ it is easy to compute $p+q$ by

$$n - \varphi(n) + 1 = pq - pq + p + q - 1 + 1 = p + q$$

and $p - q$ by

$$(p-q)^2 = p^2 + q^2 - 2pq = (p+q)^2 - 4pq = (n - \varphi(n) + 1)^2 - 4n.$$

Then,

$$\begin{aligned} p &= \frac{(p+q) + (p-q)}{2} \\ q &= \frac{(p+q) - (p-q)}{2}. \end{aligned}$$

QED

RSA — Strength (cont.)

Theorem: Computing d is equivalent to factoring n.
Sketch of Proof:

1. Given d it is easy to compute a multiple of $\varphi(n)$ by $e \cdot d - 1$. Denote it by $2^k \cdot t$, where t is odd.

2. Take a random integer r. [With a good probability (usually about half) r is a quadratic non-residue (whose orders modulo p and modulo q are divisible by different powers of 2).]

3. Compute $r_1 = r^t \pmod{n}$, and then iteratively $r_i = r_{i-1}^2 \pmod{n}$ till $r_j = 1 \pmod{n}$ for some j.

4. Then, r_{j-1} is a square root of 1.

5. We will see later that a non-trivial square root of 1 allows to compute the factorization of n by $\gcd(\cdot, n)$.

QED

RSA — Strength (cont.)

Therefore, any algorithm that computes the secret key given e, n (and possibly encrypted messages) can be converted to an algorithm to factor n.

Note: Computing m from $E(m) = m^e \bmod n$ is not known to be equivalent to factoring (since it does not require to compute d). It requires "only" to compute an e'th root, however, computing the e'th root modulo n is also a difficult problem.

The particular case of computing square roots was shown to be equivalent to factoring, but this function is not 1-1!

Signatures using RSA

Since the domain and the range of RSA are equal (Z_n), RSA can be used both for encryption and for signing.
Given the public key e_A, n_A and the secret key d_A, A signs a document m (actually signs $H(m)$) by

$$S = D_A(m) = m^{d_A} \bmod n_A,$$

and any other user can verify the signatures by checking whether

$$m \stackrel{?}{=} E_A(S) = S^{e_A} \bmod n_A.$$

Forging signatures is difficult since D_A should be computed to forge a signature. To forge a signature either d_A should be known, or an efficient algorithm to compute the e_A'th root should be known.

Rabin's RSA Variant

Reference: M. Rabin, *Digitalized Signatures and Public Key Functions as Intractable as Factoring*, Technical report MIT/LCS/TR-212, January 1979.

Rabin's RSA variant is similar to RSA but uses $e = 2$. This choice allows to prove the equivalence to factoring.

Notes:

1. Using $e = 2$ decryption is not unique, since ciphertexts can have four distinct roots.

2. There is no d such that $ed \equiv 1 \pmod{\varphi(n)}$, since $\gcd(e, \varphi(n)) = 2 \neq 1$.

Difficulty of Computing Modular Square Roots

1. It is difficult to compute modular square roots modulo n whose factorization is unknown. Computing square roots modulo n is **equivalent** to factoring n.

2. It is easy to compute square roots modulo prime numbers:

   - $p = 4k + 3$: Let α be a quadratic residue modulo p. Then

     $$\beta \equiv \alpha^{\frac{p+1}{4}} \equiv \alpha^{k+1} \pmod{p}$$

     is a square root of α:

     $$\beta^2 \equiv \alpha^{\frac{p+1}{2}} \equiv \alpha\alpha^{\frac{p-1}{2}} \equiv \alpha 1 \equiv \alpha \pmod{p}.$$

   - $p = 4k + 1$: There is a probabilistic algorithm to compute the modular square roots.

Difficulty of Computing Modular Square Roots (cont.)

3. It is easy to compute square roots modulo n whose factorization is known, by computing modulo each of the prime factors, and using the Chinese remainder theorem.

Difficulty of Computing Modular Square Roots (cont.)

Theorem: Let $n = pq$, and let $m \in Z_n^*$ be a quadratic residue. Given the four square roots of m, it is easy to factor n.

Proof: Let the square roots of m modulo p be β and $-\beta$, and let the square roots of m modulo q be γ and $-\gamma$.

Then, the four square roots of m modulo n are

$$\begin{aligned}
\alpha_{++} &: \alpha_{++} \equiv +\beta \pmod{p}, \quad \alpha_{++} \equiv +\gamma \pmod{q} \\
\alpha_{+-} &: \alpha_{+-} \equiv +\beta \pmod{p}, \quad \alpha_{+-} \equiv -\gamma \pmod{q} \\
\alpha_{-+} &: \alpha_{-+} \equiv -\beta \pmod{p}, \quad \alpha_{-+} \equiv +\gamma \pmod{q} \\
\alpha_{--} &: \alpha_{--} \equiv -\beta \pmod{p}, \quad \alpha_{--} \equiv -\gamma \pmod{q}
\end{aligned}$$

Clearly, $\alpha_{++} \equiv -\alpha_{--} \pmod{n}$ and $\alpha_{+-} \equiv -\alpha_{-+} \pmod{n}$.

We can see that $\alpha_{++} \equiv \alpha_{+-} \pmod{p}$. Thus, $\alpha_{++} - \alpha_{+-} \equiv 0 \pmod{p}$.

But, $\alpha_{++} - \alpha_{+-} \not\equiv 0 \pmod{n}$.

Therefore, $p = \gcd(\alpha_{++} - \alpha_{+-}, n)$. Similarly, $q = \gcd(\alpha_{++} + \alpha_{+-}, n)$. QED

Difficulty of Computing Modular Square Roots (cont.)

Theorem: Computing square roots modulo n is equivalent to factoring n.
Proof:
($\Leftarrow$) Given p and q anybody can compute square roots just as the signer do, by computing modulo p and modulo q.
($\Rightarrow$) Let A be an algorithm which computes square roots modulo n.
Define a probabilistic algorithm B to factor n using the algorithm A:

1. Choose a random $\alpha \in Z_n$.

2. If $\gcd(\alpha, n) > 1$ then n is factored into $\gcd(\alpha, n)$ and $\frac{n}{\gcd(\alpha,n)}$.

Difficulty of Computing Modular Square Roots (cont.)

3. If $\gcd(\alpha, n) = 1$ then $\alpha \in Z_n^*$.

   Compute $m = \alpha^2 \bmod n$ and apply Algorithm A to compute a square root β of m modulo n:

   $$\beta^2 \equiv m \equiv \alpha^2 \pmod{n}.$$

   m has **four** roots modulo n. Two of them are α and $-\alpha$. Since A does not have any information on which root α of m was chosen by B, it returns with probability half one of the roots α or $-\alpha$, and with probability half one of the other two roots.

4. If A returns a root $\beta \equiv \pm\alpha \pmod{n}$, Algorithm B restarts again from step 1.

5. Otherwise (the four square roots of m are α, $-\alpha$, β, and $-\beta$), B recovers the factors by computing $\gcd(\alpha - \beta, n)$ and $\frac{n}{\gcd(\alpha-\beta,n)}$.

6. In each step there is a probability half to find the factorization of n. After k steps the probability of failure is only 2^{-k}. QED

Rabin's RSA Variant (cont.)

Key generation:
User A chooses his keys by:

1. Randomly chooses two large prime numbers p and q of size 512 bits (154 decimal digits) at least (as in RSA).
2. Computes $n = pq$.
3. Chooses $e = 2$.
4. Publishes n as the public key, and keeps p and q secret as the secret key.

Rabin's RSA Variant (cont.)

The encryption algorithm E:
Let m be a message with several bits of known redundancy.

$$c = E_A(m) = m^2 \bmod n_A.$$

The decryption algorithm D:
User A computes the modular square root of c. He can compute modular square roots since he knows the factorization of n. He gets four roots, one of which is the message m.
In order to be able to identify m, some redundancy must be added to m before encryption (such as have some fixed value in predetermined bits).

Rabin's Signature

Signature generation:

1. Given a document m to sign, a value u of a short length (say three bits) is chosen, and m and u are concatenated to

$$c = m||u.$$

(note that the size of m should be slightly shorter than the size of n).

2. The signer A tests whether $\gcd(c, n) = 1$ and whether c is a quadratic residue modulo n (it takes $O(\log n)$ steps).

3. If the tests fail, A chooses another value u and tries again. (On average she chooses four u's till both tests succeed).

4. When both tests succeed, A computes a square root x of c: $x^2 \equiv c \pmod{n}$.

5. The signature is x

$$S(m) = x.$$

Rabin's Signature (cont.)

Signature verification:
Given m and $S(m)$, the verifier computes

$$c' = (S(m))^2 \bmod n,$$

removes the rightmost bits of c'

$$c'' = c' \gg 3,$$

and checks whether

$$c'' \stackrel{?}{=} m.$$

Rabin's Signature (cont.)

Note: Rabin's encryption and signature verification require only one modular multiplication, while using RSA about $O(\log n)$ modular multiplications are required (or 2 if e is chosen to be small $e = 3$).
The decryption and signature generation require to compute square roots (given p and q), whose complexity is similar to $O(\log n)$ modular multiplications, i.e., similar to the complexity using RSA.

Security

Breaking Rabin's variant allows the attacker to compute modular square roots. As we already proved, the ability to compute modular square roots allows to factor.

Therefore, any successful attack on Rabin's variant is as difficult as factorization (whereas RSA does not have such a proof).

A Paradox

Paradox: The proof that decryption is equivalent to factorization of n suggests an efficient method to break the cryptosystem using a chosen ciphertext attack:

1. Apply the Algorithm B described in the proof.
2. Each time Algorithm B requires to compute a square root, **request the owner of the secret key to decrypt** (or sign)!

There are several such cases in cryptography where a proof of hardness also suggests a shortcut for the attacker, but it should not be understood that all proofs lead to this phenomena. It only means that the theorem is not strong enough, or that the security criteria we use are too strong.

Rabin's RSA Variant – Revisited

In order to protect against attacks based on this paradox, some redundancy should be added to plaintexts before encryption.
Then, after decryption, the receiver can verify whether the plaintext is in the correct format. If it is not, she should ignore the decrypted plaintext.
This way, if an attacker performs this attack, he is almost always detected. The probability that he succeeds to find the factorization becomes very small.

The number of bits of redundancy should not be too small, preferably not less than 10, but also do not have to be very large, as with, say, 60 bits of redundancy, the probability of a successful attack is negligible.

Adding redundancy: setting several bits of the plaintext to be fixed, or a function of the other bits, to allow the receiver to verify existence of this redundancy.

Rabin's Signature – Revisited

For signatures we need to add some randomness, in order to decrease the probability that the signer calculates a square root of a value chosen by the attacker.

Signature generation:

1. Given a document m to sign, a random word u of a predetermined length (such as 60 bits) is chosen, and

$$c = H(m\|u)$$

   is computed by applying a public hash function H, whose range is a subset of $\{0, 1, \ldots, n_A - 1\}$. ($m\|u$ is the concatenation of m and u).

2. The signer A tests whether $\gcd(c, n) = 1$ and whether c is a quadratic residue modulo n (it takes $O(\log n)$ steps).

3. If the tests fail, A chooses another random value u and tries again. (On average he chooses four u's till both tests succeed).

Rabin's Signature – Revisited (cont.)

4. When both tests succeed, A computes a square root x of c: $x^2 \equiv c \pmod{n}$.

5. The signature is

$$S(m) = (u, x)$$

such that

$$x^2 \equiv H(m\|u) \pmod{n}.$$

Signature verification:
Given m and $S(m) = (u, x)$, the verifier checks whether

$$x^2 \stackrel{?}{\equiv} H(m\|u) \pmod{n}.$$

A Weakness of Rabin/RSA with Small Exponents

This weakness exists in Rabin's variant and in RSA with small exponents e. It exists only for encryption!
Assume that in a network, all the users use the same small e, and whose public keys differ only in the modulus n.

1. In Rabin's variant e is always considered to be $e = 2$ for all the users.

2. In RSA, if $e = 3$ or some other small odd integer.

A Weakness of Rabin/RSA with Small Exponents (cont.)

Suppose a message m has to be sent to e (or more) distinct users. To each user U, the message m is encrypted under the user's public key $E_U(m)$. The ciphertexts $C_U = E_U(m)$ are sent to the users.

Suppose an eavesdropper listens to the encrypted messages and knows C_{U_1}, $C_{U_2}, \ldots, C_{U_e}$.

A Weakness of Rabin/RSA with Small Exponents (cont.)

Then, m^e can be found by the eavesdropper by computing the unique value x modulo $\prod n_i$ which is congruent to all the C_{U_i}'s modulo n_i:

$$\begin{aligned} x &\equiv C_{U_1} \pmod{n_1} \\ x &\equiv C_{U_2} \pmod{n_2} \\ &\vdots \\ x &\equiv C_{U_e} \pmod{n_e} \end{aligned}$$

Using the Chinese remainder theorem, the unique x modulo $n_1 n_2 \cdots n_e$ can be found.
We conclude that

$$x \equiv m^e \pmod{n_1 n_2 \cdots n_e}.$$

A Weakness of Rabin/RSA with Small Exponents (cont.)

Since $m < n_i,\ \forall i \in \{1, 2, \ldots, e\}$ then $m^e < n_1 n_2 \cdots n_e$ and thus

$$x = m^e.$$

A standard (non-modular) e'th root of x can be efficiently computed, and the result is just the secret message m.

How to use RSA Correctly?

Due to the algebraic structure of RSA (and Rabin's variant), there are several undesirable properties, for example:

1. Multiplication property
2. The Jacobi symbol is not affected by encryption in RSA
3. The paradox of Rabin's variant
4. The weaknesses of low exponent RSA/Rabin

We already saw two types of partial solutions:

1. Adding redundancy
2. Adding randomness

How to use RSA Correctly? (cont.)

We now show an example where even both solutions simultaneously do not help

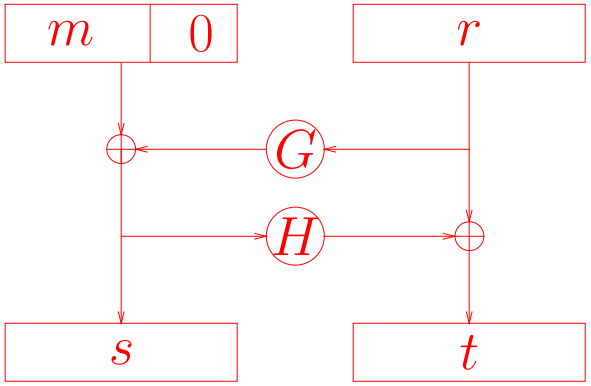
Example: The PKCS#1.5 standard:
The PKCS#1.5 involves the following steps before encryption with RSA

1. denote the plaintext by m
2. select a value r at random (with all bytes non-zero)
3. Let 0 and 2 denote bytes with the corresponding values
4. compute $m' = 0\|2\|r\|0\|m$ (where $0 \leq m' < n$)

Then, m' is encrypted by RSA, instead of m.
After several years of use of this standard, it was found that a complex chosen ciphertext attack can recover the plaintexts, if only the receiver responds with an error notice whenever the decryption fails (i.e., there are no 0 or 2 bytes in the appropriate locations of the decrypted message).

The OAEP Scheme

Due to various attacks of modes of use of RSA, it became necessary to select modes that can be proven secure under some models of threats.
OAEP (Optimal Asymmetric Encryption Padding) is one (but not the only) such attempt. It adds both redundancy and randomness to a message before encryption in the following way:



where 0 denotes zero bits, r denotes a random value, m is the plaintext ($m||0||r < n$), G and H are pseudo-random functions (such as cryptographic hash functions), and $s||t$ is the value on which the RSA encryption is performed.
The resulting encryption is: Select the random r and compute

$$c = (\text{OAEP}(m, 0, r))^e \bmod n.$$

The OAEP Scheme (cont.)

Decryption is performed by

$$(m, z, r) = \mathrm{OAEP}^{-1}(c^d \bmod n),$$

followed by verification that $z = 0$. If $z = 0$, the decrypted message is m. Otherwise, the ciphertext was forged, and the decrypted value should be ignored.

It can be shown that under some security models and some ideal selections of G and H, the resulting encryption is secure.

Similar padding schemes exist also for signatures — replacing the simple hashing of the messages described earlier in the course.

ElGamal Signature Scheme

System parameters:

1. Let p be a large prime (512 bits).
2. Let g be a generator of Z_p^*.
3. p and g can be common to all the users, or be distinct for each user.

Public and secret keys:

1. User U chooses a random secret key $X = X_U$.
2. Computes the public key $Y = Y_U = g^{-X_U} \bmod p$.

ElGamal Signature Scheme (cont.)

Signature generation: Given a message m, U signs by

1. Chooses a random r $(1 \leq r < p)$, (invertible modulo $p-1$).
2. Computes $R = g^r \bmod p$.
3. Computes $S = ((m + XR)r^{-1}) \bmod (p-1)$.
4. The signature on m is the pair (R, S).

Signature verification: Given m and an alleged signature (R, S), everybody can verify that U generated the signature by

$$Y^R R^S \stackrel{?}{=} g^m \pmod{p}.$$

ElGamal Signature Scheme (cont.)

Correctness:

$$\begin{aligned} Y^R R^S &= (g^{-X})^R (g^r)^{(m+XR)r^{-1}} \\ &= g^{-XR} g^{m+XR} = g^{-XR+m+XR} = g^m \pmod{p} \end{aligned}$$

ElGamal Signature Scheme (cont.)

Security:

1. Computing the secret key from the public key is equivalent to computing DLOG.

2. It is believed that computing the secret key using also many signed messages is as difficult as computing DLOG.

3. It is believed that signing without knowing the secret key is as difficult as computing DLOG.

4. It is very important to use random r's generated independently for each signature; otherwise the secret key might be recovered from a few signatures.

ElGamal Signature Scheme (cont.)

Advantage: In the signature generation, r, r^{-1}, R and XR can be computed in advance, before m is known. Thus, the signature generation requires only one modular multiplication in real-time (to compute S).

Schnorr's Signature Scheme

Schnorr's Signature is a variant of the ElGamal Signature.

System parameters:

1. Let p be a large prime (512 bits).
2. Let q be a smaller prime (140 bits) which divides $p-1$.
3. Let α be with order q in Z_p^*.
4. A one-way hash function $h : Z_p \times Z \rightarrow \{0, \ldots, 2^t - 1\}$, for some security parameter $t \geq 72$.
5. These parameters can be common to all the users, or be distinct for each user.

Schnorr's Signature Scheme (cont.)

Public and secret keys:

1. User U chooses a random secret key $s = s_U \in Z_q$.

2. Computes the public key $v = v_U = \alpha^{-s_U} \bmod p$.

Schnorr's Signature Scheme (cont.)

Signature generation: Given a message m, U signs by

1. Chooses a random $r \in Z_q$.
2. Computes $x = \alpha^r \bmod p$.
3. The above steps can be done in advance (preprocessing) as they do not involve knowledge of m.
4. Computes $e = h(x, m)$.
5. Computes $y = r + se \bmod q$.
6. The signature on m is the pair (e, y).

This scheme is very efficient for signing as after the preprocessing, the signer needs to perform only one modular multiplication and one modular addition, both modulo the smaller prime q.

Schnorr's Signature Scheme (cont.)

Signature verification: Given m and an alleged signature (e, y), everybody can verify that U generated the signature by computing

$$\bar{x} = \alpha^y v^e \bmod p$$

and checking whether

$$e \stackrel{?}{=} h(\bar{x}, m).$$

Schnorr's Signature Scheme (cont.)

Correctness: Exercise.
Security:

1. Computing the secret key from the public key is equivalent to computing DLOG.

2. The advantage of this scheme over ElGamal is that the corresponding authentication protocol (i.e., when e is selected at random) is zero knowledge.

3. It is believed that computing the secret key using also many signed messages is as difficult as computing DLOG.

4. It is believed that signing without knowing the secret key is as difficult as computing DLOG.

5. It is very important to use random r's generated independently for each signature; otherwise the secret key might be recovered from a few signatures.

The Digital Signature Standard (DSS)

DSS (also known as DSA: the Digital Signature Algorithm) is a US NIST standard based on Schnorr's signature. The modification was mainly done to avoid patent issues (as Schnorr's signature is patented), but the success in avoiding the patent issues is questionable.

Note: As of December 1998, RSA signatures are also approved by NIST.

The Digital Signature Standard (DSS) (cont.)

System parameters:

1. Let p be a large prime (512 bits, can be increased up to 1024 bits by multiples of 64 bits).
2. Let q be a 160-bit prime which divides $p-1$.
3. Let g be with order q in Z_p^* (select it by taking any $h \in Z_p$ and computing $g = h^{(p-1)/q}$).
4. The one-way hash function SHA-1.
5. These parameters can be common to all the users, or be distinct for each user.

The Digital Signature Standard (DSS) (cont.)

Public and secret keys:

1. User U chooses a random secret key $x = x_U \in Z_q$.

2. Computes the public key $y = y_U = g^{x_U} \bmod p$.

The Digital Signature Standard (DSS) (cont.)

Signature generation: Given a message m, U signs by

1. Chooses a random $k \in Z_q$.
2. Computes $r = (g^k \bmod p) \bmod q$.
3. The above steps can be done in advance (preprocessing) as they do not involve knowledge of m.
4. Computes $s = \left(k^{-1} \cdot (\text{SHA-1}(m) + xr)\right) \bmod q$.
5. The signature on m is the pair (r, s).

This scheme is very efficient for signing as after the preprocessing, the signer needs to perform only one modular multiplication and one modular addition, both modulo the smaller prime q.

The Digital Signature Standard (DSS) (cont.)

Signature verification: Given m and an alleged signature (r, s), everybody can verify that U generated the signature by computing

1. $w = s^{-1} \bmod q$.
2. $u1 = (\text{SHA-1}(m)w) \bmod q$.
3. $u2 = rw \bmod q$.
4. $v = ((g^{u1} y^{u2}) \bmod p) \bmod q$.

and checking whether

$$v \stackrel{?}{=} r.$$

The Digital Signature Standard (DSS) (cont.)

Correctness: Exercise.
Security: As in Schnorr's signature.

Public Key Cryptography in Practice

How Cryptography is Used in Applications

The main drawback of public key cryptography is the inherent slow speed of the public key schemes.
There are only a few schemes which are relatively faster, but they require use of huge keys, and are thus impractical.
Therefore, public key schemes are not used directly for encryption.
Instead, public key schemes are used in conjunction with secret key schemes where encryption is performed by the secret key schemes (e.g., Triple-DES) and the agreement on the keys is performed by public key distribution schemes (e.g., using RSA or Diffie-Hellman).
This is similar to the case described in the public key signature schemes, where the signature scheme does not sign the original message, but rather signs the result of a fast hash function.
Moreover, in many application even the single public key signature on a message is too cumbersome. In such cases, MACs are used, and their key is distributed in advance by a public key distribution scheme.

Recommended Key Sizes

In secret key schemes the trend changes from keys of 56–64 to keys of 128 bits. Keys of 128 bits are large enough to thwart any practical attack, as long as the cipher does not have weakness due to its design. Paranoids can use even longer keys, which are supported by various ciphers.

The situation is different in public key schemes, where considerably longer keys are required, as the keys are not uniformly selected from all the possible keys with the same length. Therefore, the number of keys is (slightly) smaller than the number of values of the same length as the keys.

However, the main reason that requires longer keys is the information inherited in the key due to the properties of the cipher.

Recommended Key Sizes (cont.)

In RSA, the public key is a product of two primes. The best known factoring algorithms are the quadratic sieve and the number field sieve whose complexities are about

$$\text{Complexity(QS)} = e^{c\sqrt{\ln n \ln\ln n}} \quad ; \quad \text{Complexity(NFS)} = e^{c(\ln n)^{1/3}(\ln\ln n)^{2/3}}$$

Due to the different constant factors (and other smaller terms) the quadratic sieve is faster when factoring up to about 129 decimal digits. The quadratic sieve algorithm was used to factor the number RSA-129, proposed by the designers of RSA in 1978 as an example of a number whose factoring will take about 40 quadrillion years. This factorization took a few months on several thousands computers over the Internet.

Over a similar computer network the NFS can factor numbers up to about 140 digits. It is expected that within a decade numbers of up to 154 digits (512 bits) will be factorable.

Therefore, all new applications should use public keys of 1024 bits. Long-term keys should have at least 2048 bits. Paranoids can use longer keys.

Recommended Key Sizes (cont.)

Breaking DLOG-based schemes requires computation of discrete logarithms. Advances in designing algorithms for computing DLOG were performed in parallel to the design of algorithms for factoring, and actually the best known algorithms for computing DLOG are variants of those used for factoring.
Nowadays, 400-bit primes moduli are still secure for DLOG-based schemes. However, it would better be that new applications use longer keys (e.g., 512 bits).
It is worth noting that many of the DLOG-based schemes can work in any group where the DLOG-problem (or the Diffie-Hellman problem) are hard. It seems that when the used group is based on elliptic curves (either an elliptic curve or an hyperelliptic curve), then the DLOG problem is significantly harder than over Z_p^*. This allows using shorter primes (or smaller groups), which gives a clear performance advantage.

Public Key Infrastructure

Public key cryptography provide a tool for secure communication between parties by letting them trust messages encrypted or signed by the **already known** public keys of the other parties.
However, no algorithmic scheme can solve the original trust problem of accepting the identity of a party that you never met.
The usual face-to-face identification is by a trusted third party (a friend) who presents the two parties to each other.
Such a presentation protocol is also required for cryptographic protocols.
The presenting party in the cryptographic environment is called a **certification authority**, or briefly a **CA**. The management of the CA's requires a **public key infrastructure (PKI)**.

Certificates

During face-to-face presentation, the presenter gives the relation between the name and the face of the presented party, together with some side information (e.g., he is a friend of the presenter).
For cryptographic use the certification authority should give the relation between the **public key** and the **identity** of his owner.
This information should be transmitted authenticated from the CA to the receiver, e.g., signed under the widely known public key of the certification authority.
Note that it is not necessary that the receiver communicate directly with the CA. Instead, the CA signs all the required information, and gives the key owner, who can then give it to anybody he wishes to communicate with, or publish widely. This, the receiver should only verify the signature of the CA, rather than to communicate with him for verifying every new key.
Such a signed information is called a **certificate**.

Certificates (cont.)

A certificate includes

1. The CA name
2. Sequential number of the certificate
3. The public key of the user
4. The identity of the user
5. Date
6. Last validation date
7. signature of the CA on all the above

Certificates (cont.)

It might happen that the secret key of some user become known to other, due to theft, factoring, or other reasons. Therefore, certification authorities maintain **Certificate Revocation Lists** (CRLs, blacklists) of canceled certificates which must not be trusted. Users can ask to add their old certificates to the blacklists if they suspect that their secret keys became known.

The last validation date field in the certificate ensures that the blacklists will not have to keep such certificates for more than a selected time, as after the last validation date, the certificates become invalid anyway, and the user should select another key instead.

The Legal Status of Digital Signatures

Several countries, including Israel, made special law to approve digital signatures for legal purposes.
Under the Israeli law, there are three kinds of digital signatures:

1. **Digital signature**: Any kind of electronic data that is added to a document to show the identity if the signer (e.g., a scanned hand-written signature at the bottom of a document). This kind has no legal status.

2. **Protected digital signature**: A digital signature that allows verification of the identity of the signer, and ensures that the signed message is original and was not modified after the signature generation.

3. **Certified digital signature**: A protected digital signature, whose key is certified by a certificate (signed by a CA).

The Legal Status of Digital Signatures (cont.)

A CA should be approved by the CA registrar, sign the certificates with a long key (at least 2048 bits in case of RSA), and satisfy many other security and financial requirements.

Certified digital signatures can be used whenever a signature is required by law (a few exceptions apply, e.g., wills), and courts accept certified digital signatures (and with the proper evidence also protected digital signatures) as valid signatures.

The X.509 Public Key Hierarchy

The X.509 standard defines a tree hierarchy of CA's. Each CA has some "parent", who signs the certifies the CA's public key. The only required widely known public key is the key of the root CA. All other public keys of CA can be verifies using certificates.

Then, a receiver verifies a certificate of another user by verifying the certificate of the CA first, and then verifying the signature of the CA on the certificate of the user. In turn, verification of the certificate of the CA is performed by verifying the certificate of his parent CA and the signature of the parent CA, and so on. Only the public key of the root CA should not be verified, as it is widely known.

The PGP Hierarchy

The drawback of the X.509 hierarchy is that every user must trust all the CA's. If a user does not trust even one CA, he cannot trust the system at all. That like "If you do not trust the government, you cannot trust your brother".
In the PGP hierarchy every user is also a CA, and users can select which CA's they trust, and which they do not trust.
As a CA a user signs certificates to his friends. His signature ensure that he recognizes the friend, and checked his identity. It does not mean that the friend is trustworthy.
Each user then asks for certificates from many other users, and collects as many as he wishes.

The PGP Hierarchy (cont.)

When a user need to prove his identity, he publishes (or sends) the certificates he collected to the other user, and the other user verify them.
The receiver can select his own trust scheme. He can decide to trust certificates signed by some CA's unconditionally, and not trust certificates signed by some other CA's. He can also decide to trust some CA with some medium trust, i.e., a certificate is only trusted if he got one (or more) additional certificates for the same key from medium trust CA's.