

COMSM2004 : Digital Signatures

B. Warinschi and N.P. Smart

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB
United Kingdom.

January 30, 2009

Outline

Generalities on Signatures

Cryptographic Hash Functions

Security of Signature Schemes

Secure RSA Signatures

DSA

Schnorr Signatures

Implicit Certificates

Signatures with Message Recovery

Recap: Public Key Encryption

Recall, the basic idea is

Message + Alice's Public Key = CipherText

CipherText + Alice's Private Key = Message

Hence, anyone with Alice's public key can send Alice a secret message.

Only Alice can decrypt the message since

- ▶ only Alice has the private key!

All they need do is look up Alice's public key in some **directory**.

Or receive her public key and a corresponding **certificate**.

Digital Signatures

Another very important public key primitive is the **digital signature**.

The idea is

Message + Alice's Private Key = Signature

Message + Signature + Alice's Public Key = YES/NO

Alice can **sign** a message using her private key.

Anyone can **verify** Alice's signature, since everyone can obtain her public key.

After verification the verifier is convinced that only Alice could have produced the signature because

- ▶ only Alice knows her private key!

Note: The above outline is a **signature scheme with appendix**.

Digital Signatures

On the last slide we described a signature scheme **with appendix**: the message is an explicit input of the verification algorithm.

Some signature schemes have the property of **message recovery**: the message is recovered from a signature.

The basic idea is

Message + **Alice's Private Key** = **Signature**

Signature + **Alice's Public Key** = **Message** or **INVALID**

Henceforth we denote a public/secret key pair (pk , sk).

A message is denoted m , the signing algorithm is denoted S , the verification algorithm is denoted V a signature is denoted s

So, we have $S(sk, m) = s$ and $V(s, m, pk) = YES/NO$. (For with appendix case.)

Digital Signatures : Services

The verification algorithm is used to determine whether or not the signature is properly constructed.

It determines whether or not the owner of the public key really produced the signature.

If s is a valid signature for m the verifier has guarantee of

- ▶ message integrity and
- ▶ message origin.

Signature schemes also provide non-repudiation - not provided by message authentication codes (MACs).

RSA Signatures

Recall RSA.

Alice picks two large primes p and q .

- ▶ p, q have around 512 bits.
- ▶ $N = pq$.

Alice also chooses an encryption exponent e with

$$\gcd(e, (p-1)(q-1)) = 1$$

Alice publishes (N, e) : The public key

Via xgcd Alice computes d such that

$$e \cdot d \equiv 1 \pmod{(p-1)(q-1)}.$$

Alice keeps (d, p, q) secret: The secret/private key

Encryption: $c = m^e \bmod N$

Decryption: $m = c^d \bmod N$

RSA Signatures

RSA can be used as a signature scheme.

- ▶ Sender applies decryption transform to generate the signature:
 $s = m^d \bmod N$.
- ▶ Receiver applies encryption transform to recover original message: $m = s^e \bmod N$.

How do we check for validity of the signature?

- ▶ If original message is natural language can verify the extracted message is also in natural language.
- ▶ Not a good idea in general!

Hence we (may) need to add **redundancy** to the message before signing.

- ▶ If the extracted message does not have the required redundancy it is not accepted.

Other Signature Schemes

RSA is by no means the only signature scheme.

Another important class of signature scheme is the **ElGamal** family based on **discrete logarithms** modulo a prime p .

- ▶ See DSA later on or Schnorr Signatures.

Other schemes of growing importance are based on discrete logarithms in other groups

- ▶ notably **elliptic curve groups** with fully exponential complexity!

Signing Long Messages

To apply RSA signing to a long message m one could

- ▶ break m into blocks m_1, m_2, \dots and
- ▶ sign each block in turn.

This is very time consuming for long messages!

Worse than this, we must add serial numbers and redundancy to each message!

- ▶ If we don't do this an attacker could delete parts of the long message without us knowing.

Luckily there is a much better way to sign long messages using RSA!

The method we have just described is a signature scheme with **message recovery**, the one we are about to see is a signature scheme **with appendix**.

Signatures Schemes with Appendix (or without Message Recovery)

Suppose we have a signature transform S that can sign $m \in \{0, 1\}^n$ using a secret key sk : $s = S(m, sk)$.

Having computed s we transmit the pair (m, s) .

The verification process V takes three inputs: message m , signature s and public key pk .

The verification process outputs a bit indicating whether or not s is a valid signature on m under public key pk .

To use this process to obtain signatures on messages of arbitrary length - as opposed to messages from $\{0, 1\}^n$ - we require the use of a **cryptographic hash function**.

Cryptographic Hash Functions

We introduce **cryptographic hash functions** to help us construct signatures for long messages.

These are functions with the following properties.

- ▶ They take **arbitrary length** bit strings as input (denote $\{0, 1\}^*$).
- ▶ They produce **fixed length** bit strings as output ($\{0, 1\}^n$).

The output of a hash function is referred to as a **fingerprint**, **message digest**, **hash code**, **hash value** or simply **hash** of its input.

There are several security properties that should be satisfied for a cryptographic hash function $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$:

- ▶ One-way property: Given $H(m)$ it should be infeasible to find **any** m' such that $H(m) = H(m')$.
- ▶ Collision resistance property: It should be infeasible to find **any** $m \neq m'$ such that $H(m) = H(m')$.

An RSA Signature Scheme with Appendix

Using a cryptographic hash function H it is possible to create a signature scheme with appendix (i.e. without message recovery) based on RSA.

Suppose we have an RSA key pair (e, N) , (d, N) such that N has n -bits.

We use a hash function $H : \{0, 1\}^* \rightarrow \{0, 1\}^n$.

To sign $m \in \{0, 1\}^*$:

- ▶ Compute $H(m)$.
- ▶ Compute signature by 'decrypting' $H(m)$: $s = H(m)^d \bmod N$.

An RSA Signature Scheme with Appendix

To verify signature s on message m :

- ▶ 'Encrypt' s to recover $H(m)' = s^e \bmod N$.
- ▶ Compute $H(m)$.
- ▶ Check whether $H(m)' = H(m)$.
- ▶ If $H(m)' = H(m)$, accept the signature. Otherwise reject.

The standardised way of doing this is slightly different (we will see it in a later lecture) but the intuition is the same.

RSA Signature: One-Way Property

The one-way property stops a cryptanalyst from cooking up a message with a given signature.

An adversary proceeds as follows.

- ▶ Chooses a random value s .
- ▶ Computes $H(m)' = s^e \bmod N$.
- ▶ Finds m such that $H(m) = H(m)'$.

It is easy to see that s is a valid signature on m in the above.

Conclusion: For our construction to be secure H must be one-way.

RSA Signature: Collision Resistance Property

We also require H to be collision resistant.

Suppose that H is not collision resistant. A **malicious signer** could proceed as follows.

- ▶ Find m and m' such that $H(m) = H(m')$.
- ▶ Sign m : $s = H(m)^d \bmod N$.
- ▶ Claim that s is really a signature on m' .

The above undermines the **non-repudiation** offered by the scheme.

Birthday Paradox

It is harder to construct collision resistant hash functions than one-way hash functions.

This is owing to the **birthday paradox**. (Actually not a paradox at all.)

How probable is it that a person in the room has a particular birthday?

How probable is it that two people in this room have the same birthday?

Experiment time!

Birthday Paradox

To find a collision of a hash function H keep computing

$$H(x_1), H(x_2), H(x_3), \dots$$

until a collision occurs.

If the function has output size of n bits then we expect to find a collision after $2^{n/2}$ iterations.

Breaking the one-way property would require 2^n iterations on average.

Hence, to achieve a security level of 80 bits we need 160 bits of output.

Hash Function Security

A **cryptographic hash function** should have the following properties.

Preimage resistant : It is hard to find a message with a given hash value.

Collision Resistant : It is hard to find two messages with the same value.

Second Preimage Resistant : Given m and $H(m)$ it is hard to find m' with $H(m') = H(m)$.

The security of a signature scheme depends on both the security of the underlying public key scheme **and** the security of the hash function used in the construction.

Avalanche Effect

A basic **design principle** when designing hash functions is that the hash function should produce an **avalanche effect**.

In other words a small change in the input produces a **large** and **unpredictable** change in the output.

This is needed so that a signature on a cheque for **30 pounds** cannot be altered into a signature on a cheque for **30000 pounds**.

Avalanche Effect : Examples

MD5("The quick brown fox jumps over the lazy dog") =
9e107d9d372bb6826bd81d3542a419d6

MD5("The quick brown fox jumps over the lazy cog") =
1055d3e698d289f2af8663725127bd4b

SHA1("The quick brown fox jumps over the lazy dog") =
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

SHA1("The quick brown fox jumps over the lazy cog") =
de9f2c7fd25e1b3afad3e85a0bd17d9b100db4b3

Hash Functions in Practice

To be collision resistant, a hash-code should be at least 128 bits long and

- ▶ preferably 160 bits.

Several hash functions are widely used, they are all iterative in nature.

The **Secure Hash Algorithm - 1 (SHA-1)** is a US and ISO standard with 160 bit outputs. (Recent attacks are theoretical only.)

In the past **MD4** and **MD5** were very popular but it has been superseded by SHA-1.

In August 2004 a method published to find collisions in MD4 and MD5.

Also a reduced version of SHA-1 was “broken” at the same conference.

The MD4 Family

MD4, MD5, SHA-1 and RIPEMD-160 are all in the same 'family'.

MD4

- ▶ 3 rounds of 16 steps, output bit-length is 128

MD5

- ▶ 4 rounds of 16 steps, output bit-length is 128

SHA-1

- ▶ 4 rounds of 20 steps, output bit-length is 160

RIPEMD-160

- ▶ 5 rounds of 16 steps, output bit-length is 160

SHA-2

- ▶ Produces various output bit-lengths: 256, 384 and 512.
- ▶ Number of rounds and steps vary depending on the bit-length.

MD4

We will discuss MD4 in some detail; the others in the family are variants of this.

There are three functions of three variables

- ▶ $f(u,v,w)$
- ▶ $g(u,v,w)$
- ▶ $h(u,v,w)$

each consisting of logical bitwise operations (and, or, etc.).

We have a current **hash state** (H_1, H_2, H_3, H_4) of 32-bit values initialised with a fixed **IV** (**initial vector**).

There are various fixed constants: y_j, z_i and s_i .

MD4

The data stream is loaded 16 words at a time into $X[j]$, $0 \leq j < 16$.

The following steps are then executed.

- ▶ $(A, B, C, D) = (H_1, H_2, H_3, H_4)$
- ▶ Execute Round 1
- ▶ Execute Round 2
- ▶ Execute Round 3
- ▶ $(H_1, H_2, H_3, H_4) = (H_1 + A, H_2 + B, H_3 + C, H_4 + D)$

After all data has been read in, the output is the concatenation of H_1, H_2, H_3, H_4 .

We will now see what happens during the rounds.

MD4

Round 1 For $j = 0$ to 15 do

- ▶ $t = A + f(B, C, D) + X[z_j] + y_j$
- ▶ $(A, B, C, D) = (D, t \ll s_j, B, C)$

Round 2 For $j = 16$ to 31 do

- ▶ $t = A + g(B, C, D) + X[z_j] + y_j$
- ▶ $(A, B, C, D) = (D, t \ll s_j, B, C)$

Round 3 For $j = 32$ to 47 do

- ▶ $t = A + h(B, C, D) + X[z_j] + y_j$
- ▶ $(A, B, C, D) = (D, t \ll s_j, B, C)$

The symbol \ll denotes bit wise rotate in the above.

Merkle-Damgård Construction

Most fully-fledged hash functions are constructed from **compression functions** as described below.

Let f denote a collision resistant compression function from $n + t$ bits to n bits.

To construct a collision resistant function that takes arbitrary length input we proceed as follows.

- ▶ Divide the input into t -bit blocks m_1, \dots, m_l .
- ▶ Set h_0 to be a fixed n bit block.
- ▶ Define $h_i = f(h_{i-1} || m_i)$ for $i = 1, \dots, l$.
- ▶ Output h_l .

Often used with **length strengthening**: one adds an extra block to the message to encode its length.

Problems were found in this construction in 2004/2005, so people are now working on new ideas.

Hash Functions and MACs

A collision resistant cryptographic hash function can be used as the basis of a **message authentication code** (MAC).

This is like a digital signature except that it is secret key based.

- ▶ MACS are used as part of larger protocols for integrity checking.
- ▶ They do not offer non-repudiation as either party who knows the secret key could have generated the MAC.

Note: If a MAC has an n bit key and an n bit output then breaking it should require 2^n operations.

Hash Functions and MACs

One possibility is to concatenate the key with the message and then apply a hash-function H .

Suppose the hash code output is the MAC

$$MAC = H(k||M).$$

Suppose H is computed via the non length-strengthened Merkle-Damgård construction.

Then, given MAC and M one can compute the MAC on the message $M||N$ for any N as follows.

$$H(k||M||N) = H(MAC||N)$$

A similar attack can be mounted on the strengthened version.

Hash Functions and MACs

One possibility is to concatenate the key with the message and then apply a hash-function.

Suppose the hash code output is the MAC

$$MAC = H(M||k).$$

This is a very bad idea because of a birthday attack operating in

$$O(2^{n/2})$$

operations.

Hash Functions and MACs

The most widely adopted use of a hash in a MAC is **HMAC**

HMAC is (believed to be) secure and it works as follows.

$$MAC = H(k||p_1||H(k||p_2||M))$$

where

- ▶ k is the MAC's key, say of length 128 bits.
- ▶ m is the message we wish to MAC.
- ▶ p_1 and p_2 are padding strings, say of length 384 bits.

Another Use of Hash Functions

Hash functions are simply a special type of **manipulation detection code** (MDC).

A hash function can be used to protect the integrity of a large file as follows.

- ▶ The hash of the file is computed.
- ▶ The hash value is kept in a physically secure place, on a floppy disk in a safe for example.
- ▶ To test whether or not the file has been manipulated one compares its hash value with that on disk.

Clearly collision resistance is important in the above context!

Hash Functions and Block Ciphers

A hash function can be constructed from a block ciphers E .

There are a number of ways of doing this.

- ▶ All methods make use of a constant initial value IV .
- ▶ Some use a function g which maps n -bit inputs to keys.
- ▶ We pad the message into blocks x_0, x_1, \dots, x_t .
- ▶ The hash value is the final value of H_i in the following iteration.

Hash Functions and Block Ciphers

The general construction works as follows.

- ▶ $H_0 = IV$
- ▶ $H_i = f(x_i, H_{i-1})$

Where do the functions we are using come from?

Matyas-Meyer-Oseas Hash

- ▶ $f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i.$

Davies-Meyer Hash

- ▶ $f(x_i, H_{i-1}) = E_{x_i}(H_{i-1}) \oplus H_{i-1}.$

Miyaguchi-Preneel Hash

- ▶ $f(x_i, H_{i-1}) = E_{g(H_{i-1})}(x_i) \oplus x_i \oplus H_{i-1}.$

Signature Scheme Security

What does it mean for a signature scheme to be secure?

- ▶ **Note:** We don't care about recovering a message, since the message is public.

Like standard signatures we worry about forgery.

- ▶ There are two types of forgery:
 - ▶ **selective forgery** and
 - ▶ **existential forgery**.

Signature Scheme Security

Selective Forgery :

- ▶ Clearly we require that an attacker should not be able to produce a message, signature pair on a message of their choice.
- ▶ This is considered to be a weak notion of security.

Existential Forgery :

- ▶ A scheme is **existentially unforgeable** if, no matter how many message, signature pairs an adversary sees, it cannot produce a signature on **any** other message.

Signature Scheme Security

We have just seen some **adversarial goals**; to form a complete definition we also need **attack models**.

Passive Attack :

- ▶ Attacker obtains a public key and some message, signature pairs produced using the public key.

Adaptive Chosen Message Attack :

- ▶ Attacker can obtain signatures on messages of its choosing.
- ▶ It can choose the messages based on what it has already seen - hence **adaptive**.
- ▶ It's job is to produce a signature on a new message.

Accepted definition of security :

- ▶ A signature scheme is deemed secure if it resists **existential forgery** under an **adaptive chosen message attack**.

RSA Signatures

So, how do we turn a primitive like RSA into a signature scheme?

We could use a hash function which produces 1024 bit outputs. This is often called RSA-FDH (**full-domain hash** where domain should be $(\mathbb{Z}/n\mathbb{Z})^*$).

- ▶ It is hard to create such hash functions in practice.
- ▶ If we assume such hash functions exist and model them by **random oracles** (see later) then one can show that RSA-FDH is secure in the above sense.

Another provably secure variant (in the random oracle model) of RSA is **RSA-PSS** (**probabilistic signature scheme**).

Details of RSA-PSS may be found in the text book for the course.

RSA Signatures

A common way of using RSA as a signature algorithm is that which is defined in the PKCS-1 standard.

- ▶ This has a method of padding for both **message recovery** and **with appendix** variants.
- ▶ Very old and not known to be **provably secure** though.

Suppose our RSA function f is on n bits, e.g. $n = 1024$.

Let k denote the number of octets in n , i.e. $k = n/8 = 128$.

We present a simplified version of PKCS-1, without ASN-1 padding.

PKCS-1

To sign a message m using RSA as in PKCS-1 one proceeds as follows.

Hash m to obtain an octet string D whose length should be less than or equal to $k - 11$ in octets.

Form an octet string EB of k octets by concatenating

$$00\|BT\|PS\|00\|D.$$

Usually one chooses

- ▶ $BT = 01$
- ▶ $PS = FF\|FF\|\dots\|FF.$

Convert EB to an integer m in the obvious way (little octetian format).

The signature is then $f(m)$.

Digital Signature Algorithm

We have one digital signature scheme (RSA) why do we need another one?

- ▶ What if someone breaks the RSA algorithm?
- ▶ What if factoring is easy?

We also have DSA (**digital signature algorithm**).

- ▶ Sometimes referred to as DSS (**digital signature standard**).

The **elliptic curve** variants of DSA (**ECDSA**) run very fast and have smaller footprints and key sizes.

DSA is based on the difficulty of the **discrete logarithm problem** (DLP) in the group $GF(p)^* = \mathbb{F}_p^*$.

DSA

A DSA signature consists of two 160-bit blocks r and s .

r is a function of a 160-bit random number k

- ▶ which is different for every message (like a session key).

s is a function of

- ▶ the message,
- ▶ the signers private key x and
- ▶ r .

The signature has a 2^{-160} probability of being forged i.e. if you write down random r and s then it has probability 2^{-160} of being a valid signature for a message m .

DSA : Domain Parameters

The following are public.

Choose a 160 bit prime q , and a large prime p where

- ▶ p has 512-1024 bits and
- ▶ q divides $p - 1$.

Calculate

$$g = h^{(p-1)/q}$$

where h is a random integer less than p and $g > 1$ - choose values for h until this is so.

g is an element of order q in $GF(p)^*$ i.e.

$$g \neq 1 \pmod{p} \text{ and } g^q = 1 \pmod{p}.$$

DSA : Key Setup

Each user generates a secret signing key x at random and such that

- ▶ $0 < x < q$.

Public key is y where

$$y = g^x \pmod{p}.$$

The public key y is bound to its owner using a **certificate**

- ▶ e.g. an X.509 certificate.

DSA : Signing

To sign a message M the signer proceeds as follows.

- ▶ Signer computes one-way hash $m = H(M)$.
- ▶ Signer chooses a random ephemeral key: $0 < k < q$.
- ▶ Signer computes $r = (g^k \pmod{p}) \pmod{q}$.
- ▶ Finally, signer computes

$$s = (m + xr)/k \pmod{q}.$$

The signature on M is the pair (r, s) .

DSA : Verification

To verify a signature (r, s) on a message M under public key y , the verifier proceeds as follows.

The verifier computes the following.

- ▶ $m = H(M)$
- ▶ $a = m/s \pmod{q}$
- ▶ $b = r/s \pmod{q}$

The verifier accepts the signature if and only if $v = r$ where

$$v = (g^a y^b \pmod{p}) \pmod{q}.$$

DSA : Small Example

Domain parameters: $q = 13$, $p = 4q + 1 = 53$, $g = 16$

Private key: $x = 3$

Public key: $y = g^x \pmod{p} = 16^3 \pmod{53} = 15$

Signature:

- ▶ Hash of message $m = H(M) = 5$
- ▶ Ephemeral key $k = 2$
- ▶ $r = (g^k \pmod{p}) \pmod{q} = 5$
- ▶ $s = (m + xr)/k \pmod{q} = 10$

Verification:

- ▶ $a = m/s \pmod{q} = 7$ ($m = H(M)$)
- ▶ $b = r/s \pmod{q} = 7$
- ▶ $v = (g^a y^b \pmod{p}) \pmod{q} = 5$

Note: $v = r$ hence signature is verified.

DSA Security

The above algorithm uses the subgroup of \mathbb{F}_p^* of order q which is generated by g .

Hence the DLOG problem really is in the cyclic group $G = \langle g \rangle$ of order q .

For security we require

- ▶ $p > 2^{1024}$ to avoid attacks using the **Number Field Sieve** (NFS) and
- ▶ $q > 2^{160}$ to avoid attacks using **BSGS/Pollard's Rho**.

DSA Security

In the light of the above, to achieve 80 bits of security we need to operate on integers of 1024 bits in length (owing to NFS attack).

This makes DSA even slower than RSA, since the DSA operations are more complicated than those for RSA.

Would it not be good if we could use another group of size around 2^{160}

- ▶ that is not susceptible to NFS style attacks and
- ▶ that provided fast arithmetic.

Generalised DSA

We can generalise DSA to an arbitrary finite Abelian group in which the DLOG problem is hard.

We write $G = \langle g \rangle$ for a cyclic group generated by g .

- ▶ Assume g has prime order $q > 2^{160}$.
- ▶ Assume that the DLOG problem with respect to g is hard.
- ▶ Assume we have a public function f

$$f : \langle g \rangle \longrightarrow \mathbb{F}_q^*.$$

Each user generates a secret signing key x at random and such that

- ▶ $0 < x < q$.

Public key is y where

$$y = g^x.$$

Generalised DSA : Signing

To sign a message M the signer proceeds as follows.

- ▶ Signer computes one-way hash $m = H(M)$.
- ▶ Signer chooses a random ephemeral key: $0 < k < q$.
- ▶ Signer computes $r = f(g^k)$.
- ▶ Finally, signer computes

$$s = (m + xr)/k \pmod{q}.$$

The signature on M is the pair (r, s) .

Generalised DSA : Verification

To verify a signature (r, s) on a message M under public key y , the verifier proceeds as follows.

The verifier computes the following.

- ▶ $m = H(M)$
- ▶ $a = m/s \pmod{q}$
- ▶ $b = r/s \pmod{q}$

The verifier accepts the signature if and only if $v = r$ where

$$v = f(g^a y^b).$$

ECDSA

Elliptic Curve DSA (**ECDSA**) is a special case of the generalised DSA scheme that we have just seen.

As a group G it uses the points on some **elliptic curve**.

Given the state of the art the **elliptic curve discrete logarithm problem** (ECDLP) has full exponential complexity which gives a great deal of security.

- ▶ Keys sizes can be very small.
- ▶ This makes the system much faster.

We can use ECDLP for Diffie-Hellman key exchange and ElGamal encryption.

Elliptic curve cryptography (ECC) is becoming increasingly important in constrained devices since it uses less silicon, code, bandwidth, time etc.

Schnorr Signatures

An important DLP based signature scheme is that of Schnorr.

- ▶ It occurs in many ZK proofs and as parts of other protocols.
- ▶ It is the simplest among the DLP based schemes that are **provably secure**.
- ▶ The signing and verifying operations are simpler than those for DSA.

Take a group $G = \langle g \rangle$ of prime order q in which the DLP is hard.

Each user generates a secret signing key x at random and such that

- ▶ $0 < x < q$.

Public key is $y = g^x$.

Are you noticing a pattern here?

Schnorr Signatures : Signing

To sign a message M the signer proceeds as follows.

- ▶ Signer chooses a random **ephemeral key**: $0 < k < q$.
- ▶ Signer computes $r = g^k$.
- ▶ Signer computes one-way hash $m = H(r||M)$.
- ▶ Finally, signer computes

$$s = (k + mx) \pmod{q}.$$

The signature on M is the pair (m, s) .

Schnorr Signatures : Verification

To verify a signature (m, s) on a message M under public key y , the verifier proceeds as follows.

The verifier computes

$$r' = g^s y^{-m}.$$

If the signature is valid we have

$$r' = g^{k+mx} \cdot g^{-xm} = g^k.$$

So, the verifier accepts signature if and only if

$$m = H(r' || M).$$

Making Things Smaller

We now investigate how to make messages smaller.

We describe two such ways:

- ▶ **implicit certificates** and
- ▶ **signatures with message recovery**.

Sizes of Standard Certificates

Recall what a standard certificate looks like:

$$X||Y.$$

where

- ▶ X = (user's details, user's public key)
- ▶ Y = CA's signature on X

This holds whether we use PKIX, SPKI or PGP.

The user's details contains things like

- ▶ name
- ▶ time
- ▶ authorisation
- ▶ ...

Standard Certificate Sizes

Ignoring the size of the user's details, this gives something quite big, for example

	RSA	DSA	ECDSA
User's key	1024	1024	160
CA's signature	2048	400	400

This assumes for RSA/DSA keys one uses

- ▶ 1024 bits (160 bit subgroup for DSA) for user keys and
- ▶ 2048 bits (200 bit subgroup for DSA) for CA keys.

And for ECDSA one uses

- ▶ 160 bits for user keys and
- ▶ 200 bits for CA keys.

The main question is: can these be made smaller?

Implicit Certificates

We cannot reduce the size of the user's details but we could possibly reduce the size of the other components.

An implicit certificate looks like

$$X||Y$$

where

- ▶ X = user's details and
- ▶ Y = the actual implicit certificate on X .

From Y we can

- ▶ recover the public key of the user and gain
- ▶ implicit assurance that the certificate was issued by the CA.

For a DSA type signatures Y has approximately 1024 bits.

For ECDSA type signatures Y has size approx 160 bits

We shall describe a DSA based version here.

Implicit Certificates

System Setup

The CA chooses a group \mathbb{G} of order n and an element $P \in \mathbb{G}$. The CA chooses a private key c from $\{1, \dots, n\}$ and computes the public key

$$Q = P^c.$$

Certificate Request

For Alice to request a certificate and the public key associated to the information ID (user's details on the previous slides) she proceeds as follows.

Alice chooses an ephemeral secret key t from $\{1, \dots, n\}$ and computes an ephemeral public key

$$R = P^t.$$

Alice sends R and ID to the CA.

Implicit Certificates

Processing of the Request

The CA checks that he wants to link ID with Alice. The CA then picks another random number k from $\{1, \dots, n\}$ and computes

$$G = P^k R = P^k P^t = P^{k+t}.$$

Next it computes

$$s = cH(ID||G) + k \pmod{n}.$$

The CA sends back to Alice the pair

$$(G, s).$$

The implicit certificate is the pair

$$(ID, G).$$

Implicit Certificates

We now need to verify that

- ▶ Alice can recover a valid public/private key pair and that
- ▶ any other user can recover Alice's public key from the implicit certificate.

Key Recovery for Alice

Alice knows the following information

$$t, s, R = P^t.$$

From this she can recover her private key

$$a = t + s \pmod{n}.$$

Her public key is

$$P^a = P^{t+s} = P^t P^s = RP^s.$$

Implicit Certificates

Key Recovery for Users

Since s and R are public a user can recover Alice's public key from

$$RP^s.$$

But this says nothing about the linkage between the CA, Alice's public key and the ID information.

Implicit Key Recovery for Users

Instead the user recovers the public key from the implicit certificate

$$(ID, G)$$

and the CA's public key

$$Q$$

as follows.

$$Q^{H(ID||G)} G = P^{cH(ID||G)} P^{k+t} = P^{s+t} = P^a$$

Implicit Certificates

As soon as the users sees Alice's key used **in action** he knows implicitly that it must have been issued by the CA

- ▶ since otherwise Alice's signature would not verify correctly.

There are a number of problems with the above system which means that implicit certificates are not used much in real life.

- ▶ What do you do if the CA's key is compromised?
 - ▶ Usually you pick a new CA key and re-certify users keys.
 - ▶ You cannot do this since users' public keys are chosen interactively during the certification process.
- ▶ Implicit certificates require the CA and users to work at the same security level.
 - ▶ This is not considered good practice.

However, for small bandwidth devices they can offer a suitable solution.

Signatures with Message Recovery

What happens when we want to sign a message which is itself quite short.

- ▶ The signature could be longer than the message.

RSA can be used either as a scheme with appendix or without. We would like a DLOG scheme with this property.

The DLOG based schemes we have seen are all signatures with appendix:

- ▶ the signature is appended to the message.

We would like to remove this but still maintain security.

Redundancy Function

All signature schemes with message recovery require a redundancy function R . This maps messages over to the data which is actually signed.

A redundancy function acts rather like a hash function in standard signatures.

- ▶ But a redundancy function must be **easy to invert!**

We shall take R to be the function

$$R : \begin{cases} \{0, 1\}^{n/2} & \longrightarrow \{0, 1\}^n \\ m & \longrightarrow m || m \end{cases}$$

See the **HAC**, **Section 11.2.3** for more discussion of redundancy functions.

Nyberg-Rueppel Signature Scheme

This is a signature scheme with message recover, based on the DLOG problem in some group G .

- ▶ We shall use the integers modulo p , i.e. $G = \mathbb{F}_p^*$.
- ▶ Assume a large prime q divides $p - 1$ as usual.
- ▶ Assume g is a generator of the subgroup of order q .

Each user generates a secret signing key x at random and such that

- ▶ $0 < x < q$.

Public key is $y = g^x$.

Nyberg-Rueppel: Signing

Nyberg-Rueppel Signatures are produced by Alice, the owner of (y, x) , as follows.

- ▶ Select a random k from $0 < k < q$ and compute

$$r = g^k \pmod{p}$$

- ▶ Compute $e = R(m)r \pmod{p}$.
- ▶ Compute $s = xe + k \pmod{q}$.

The signature is then (e, s) .

From this pair, which is a group element and an integer modulo q ,

- ▶ we need to verify the signature comes from Alice and to
- ▶ recover the message m from the pair (e, s) .

Nyberg-Rueppel: Verification

Bob is given the pair (e, s) and Alice's public key $y = g^x$. Bob computes

$$u_1 = g^s y^{-e} = g^{s-ex} = g^k = r \pmod{p}.$$

Bob now computes

$$u_2 = e/u_1 \pmod{p}.$$

Bob then verifies that u_2 lies in the range of the redundancy function, in our example we must have

$$u_2 = R(m) = m||m.$$

Now, Bob can recover the message m .

Size of Nyberg-Rueppel Signatures

Since we require one group element and one integer modulo q , the size can look quite large if one uses standard DLOGs.

If one uses standard DLOGs then one really requires

- ▶ $1024 + 160 = 1624$ bits.

If one uses an ECC variant then one can get away with

- ▶ $160 + 160 = 320$ bits.