

CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 14
March 5, 2013

Using Digital Signatures

- Review: digital signatures
- Signed encrypted messages

Practical Signature Algorithms

- Digital signature algorithm (DSA)
- Common hash functions
- MD5

Digital Signatures with Special Properties

- Blind signatures

Using Digital Signatures

Asymmetric digital signatures

Recall our formal definition of an asymmetric signature scheme.

Let \mathcal{M} be a *message space* and \mathcal{S} a *signature space*.

A *signature scheme* consists of a private *signing key* d , a public *verification key* e , a *signature function* $S_d : \mathcal{M} \rightarrow \mathcal{S}$, and a *verification predicate* $V_e \subseteq \mathcal{M} \times \mathcal{S}$.¹

A *signed message* is a pair $(m, s) \in \mathcal{M} \times \mathcal{S}$. A signed message is *valid* if $V_e(m, s)$ is true, and we say that (m, s) is *signed with* e .

¹As with RSA, we denote the private component of the key pair by the letter d and the public component by the letter e , although they no longer have same mnemonic significance.

Fundamental property of a signature scheme

Basic requirement:

The signing function always produces a valid signature, that is,

$$V_e(m, S_d(m)) \quad (1)$$

is true for all $m \in \mathcal{M}$.

Signing message digests

The usual way to use a digital signature scheme such as RSA is to sign a *digest* of the message rather than sign m itself.

A message digest function h , also called a *cryptographic hash function* or a *fingerprint function*, maps long strings to short random-looking strings.

- ▶ To sign a message m , Alice computes $S_d(m) = D_d(h(m))$.
- ▶ To verify the signature s , Bob checks that $h(m) = E_e(s)$.

Signed encrypted messages

One often wants to encrypt messages for privacy and sign them for integrity and authenticity.

Let Alice have cryptosystem (E, D) and signature system (S, V) . Some possibilities for encrypting and signing a message m :

1. Alice **signs** the **encrypted** message and sends the pair $(E(m), S(E(m)))$.
2. Alice **encrypts** the **signed** message and sends the result $E(m \circ S(m))$. Here we assume a standard way of representing the ordered pair $(m, S(m))$ as a string, which we denote by $m \circ S(m)$.
3. Alice separately **encrypts** and **signs** message and sends the pair $(E(m), S(m))$.

Weakness of encrypt-and-sign

Note that method 3, sending the pair $(E(m), S(m))$, is quite problematic since *signature functions make no guarantee of privacy*.

We can construct a signature scheme (S', V') in which the plaintext message is part of the signature itself.

If (S', V') is used as the signature scheme in method 3, there is no privacy, for the plaintext message can be read directly from the signature.

A forgery-resistant signature scheme with no privacy

Example: Let (S, V) be an RSA signature scheme. Define

$$S'(m) = m \circ S(m) ;$$

$$V'(m, s) = \exists t (s = m \circ t \wedge V(m, t)) .$$

Fact

(S', V') is at least as secure as (S, V) .

A forgery-resistant signature scheme with no privacy

Example: Let (S, V) be an RSA signature scheme. Define

$$S'(m) = m \circ S(m) ;$$

$$V'(m, s) = \exists t (s = m \circ t \wedge V(m, t)) .$$

Fact

(S', V') is at least as secure as (S, V) .

Why?

A forgery-resistant signature scheme with no privacy

Example: Let (S, V) be an RSA signature scheme. Define

$$S'(m) = m \circ S(m) ;$$

$$V'(m, s) = \exists t (s = m \circ t \wedge V(m, t)) .$$

Fact

(S', V') is at least as secure as (S, V) .

Why? Suppose a forger produces a valid signed message (m, s) in (S', V') , so $s = m \circ t$ for some t and $V(m, t)$ holds..

Then (m, t) is a valid signed message in (S, V) .

Encrypt or sign first?

Method 1 (encrypt first) allows Eve to verify that the signed message was sent by Alice, even though Eve cannot read it. Whether or not this is desirable is application-dependent.

Method 2 (sign first) forces Bob to decrypt a bogus message before discovering that it wasn't sent by Alice.

Subtleties emerge when cryptographic protocols are combined, even in a simple case like this where it is only desired to combine privacy with authentication.

Encrypt or sign first?

Method 1 (encrypt first) allows Eve to verify that the signed message was sent by Alice, even though Eve cannot read it. Whether or not this is desirable is application-dependent.

Method 2 (sign first) forces Bob to decrypt a bogus message before discovering that it wasn't sent by Alice.

Subtleties emerge when cryptographic protocols are combined, even in a simple case like this where it is only desired to combine privacy with authentication.

Think about the pros and cons of other possibilities, such as sign-encrypt-sign, i.e., $(E(m \circ S(m)), S(E(m \circ S(m))))$.

Practical Signature Algorithms

Digital signature standard

The commonly-used Digital Signature Algorithm (DSA) is a variant of ElGamal signatures. Also called the Digital Signature Standard (DSS), it is described in U.S. Federal Information Processing Standard FIPS 186–3.²

It uses two primes: p , which is 1024 bits long,³ and q , which is 160 bits long and satisfies $q \mid (p - 1)$. Here's how to find them: Choose q first, then search for z such that $zq + 1$ is prime and of the right length. Choose $p = zq + 1$ for such a z .

²Available at http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

³The original standard specified that the length L of p should be a multiple of 64 lying between 512 and 1024, and the length N of q should be 160. Revision 2, Change Notice 1 increased L to 1024. Revision 3 allows four (L, N) pairs: (1024, 160), (2048, 224), (2048, 256), (3072, 256).

DSA key generation

Given primes p and q of the right lengths such that $q \mid (p - 1)$, here's how to generate a DSA key.

- ▶ Let $g = h^{(p-1)/q} \bmod p$ for any $h \in \mathbf{Z}_p^*$ for which $g > 1$.
This ensures that $g \in \mathbf{Z}_p^*$ is a non-trivial q^{th} root of unity modulo p .
- ▶ Let $x \in \mathbf{Z}_q^*$.
- ▶ Let $a = g^x \bmod p$.

Private signing key: (p, q, g, x) .

Public verification key: (p, q, g, a) .

DSA signing and verification

Here's how signing and verification work:

To sign m :

1. Choose random $y \in \mathbf{Z}_q^*$.
2. Compute $b = (g^y \bmod p) \bmod q$.
3. Compute $c = (m + xb)y^{-1} \bmod q$.
4. Output signature $s = (b, c)$.

To verify (m, s) , where $s = (b, c)$:

1. Verify that $b, c \in \mathbf{Z}_q^*$; reject if not.
2. Compute $u_1 = mc^{-1} \bmod q$.
3. Compute $u_2 = bc^{-1} \bmod q$.
4. Compute $v = (g^{u_1} a^{u_2} \bmod p) \bmod q$.
5. Check $v = b$.

Why DSA works

To see why this works, note that since $g^q \equiv 1 \pmod{p}$, then

$$r \equiv s \pmod{q} \quad \text{implies} \quad g^r \equiv g^s \pmod{p}.$$

This follows from the fact that g is a q^{th} root of unity modulo p , so $g^{r+uq} \equiv g^r(g^q)^u \equiv g^r \pmod{p}$ for any u .

Hence,

$$g^{u_1} a^{u_2} \equiv g^{mc^{-1} + xbc^{-1}} \equiv g^y \pmod{p}. \quad (2)$$

$$g^{u_1} a^{u_2} \pmod{p} = g^y \pmod{p} \quad (3)$$

$$v = (g^{u_1} a^{u_2} \pmod{p}) \pmod{q} = (g^y \pmod{p}) \pmod{q} = b$$

as desired. (Notice the shift between *equivalence* modulo p in equation 2 and *equality of remainders* modulo p in equation 3.)

Double remaindering

DSA uses the technique of computing a number modulo p and then modulo q .

Call this function $f_{p,q}(n) = (n \bmod p) \bmod q$.

$f_{p,q}(n)$ is not the same as $n \bmod r$ for any modulus r , nor is it the same as $f_{q,p}(n) = (n \bmod q) \bmod p$.

Example mod 29 mod 7

To understand better what's going on, let's look at an example. Take $p = 29$ and $q = 7$. Then $7|(29 - 1)$, so this is a valid DSA prime pair. The table below lists the first 29 values of $y = f_{29,7}(n)$:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
y	0	1	2	3	4	5	6	0	1	2	3	4	5	6	
n	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
y	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0

The sequence of function values repeats after this point with a period of length 29. Note that it both begins and ends with 0, so there is a double 0 every 29 values. That behavior cannot occur modulo any number r . That behavior is also different from $f_{7,29}(n)$, which is equal to $n \bmod 7$ and has period 7. (Why?)

Common hash function

Many cryptographic hash functions are currently in use.

For example, the openssl library includes implementations of MD2, MD4, MD5, MDC2, RIPEMD, SHA, SHA-1, SHA-256, SHA-384, and SHA-512.

The SHA-xxx methods are recommended for new applications, but these other functions are also in widespread use.

SHA-1

The revised Secure Hash Algorithm (SHA-1) is one of five algorithms described in U. S. Federal Information Processing Standard FIPS PUB 180-4 (Secure Hash Standard).⁴ It states,

“Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits).”

SHA-1 produces a 160-bit message digest. The other algorithms in the SHA-xxx family produce longer message digests.

⁴Available at <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

SHA-1 broken

SHA-1 was broken in 2005 by Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu: “Finding Collisions in the Full SHA-1”. *CRYPTO 2005*: 17-36. Wang and Yu did their work at Shandong University; Yin is listed on the paper as an independent security consultant in Greenwich, CT.

On Nov. 2, 2007, NIST announced a public competition for a replacement algorithm to be known as SHA-3.

The winner, an algorithm named Keccak, was announced on October 2, 2012. It is currently in the process of standardization.⁵

⁵See <http://csrc.nist.gov/groups/ST/hash/sha-3/index.html>

MD5 overview

MD5 is an older algorithm (1992) devised by Rivest. We present an overview of it here.

It generates a 128-bit message digest from an input message of any length. It is built from a basic block function

$$g : 128\text{-bit} \times 512\text{-bit} \rightarrow 128\text{-bit}.$$

The MD5 hash function h is obtained as follows:

- ▶ The original message is padded to length a multiple of 512.
- ▶ The result m is split into a sequence of 512-bit blocks m_1, m_2, \dots, m_k .
- ▶ h is computed by chaining g on the first argument.

We next look at these steps in greater detail.

MD5 padding

As with block encryption, it is important that the padding function be one-to-one, but for a different reason.

For encryption, the one-to-one property is what allows unique decryption.

For a hash function, it prevents there from being trivial colliding pairs.

For example, if the last partial block is simply padded with 0's, then all prefixes of the last message block will become the same after padding and will therefore collide with each other.

MD5 chaining

The function h can be regarded as a state machine, where the states are 128-bit strings and the inputs to the machine are 512-bit blocks.

The machine starts in state s_0 , specified by an initialization vector IV .

Each input block m_i takes the machine from state s_{i-1} to new state $s_i = g(s_{i-1}, m_i)$.

The last state s_k is the output of h , that is,

$$h(m_1 m_2 \dots m_{k-1} m_k) = g(g(\dots g(g(IV, m_1), m_2) \dots, m_{k-1}), m_k).$$

MD5 block function

The block function $g(s, b)$ is built from a scrambling function $g'(s, b)$ that regards s and b as sequences of 32-bit words and returns four 32-bit words as its result.

Suppose $s = s_1s_2s_3s_4$ and $g'(s, b) = s'_1s'_2s'_3s'_4$.

We define

$$g(s, b) = (s_1 + s'_1) \cdot (s_2 + s'_2) \cdot (s_3 + s'_3) \cdot (s_4 + s'_4),$$

where “+” means addition modulo 2^{32} and “.” is concatenation of the representations of integers as 32-bit binary strings.

MD5 scrambling function

The computation of the scrambling function $g'(s, b)$ consists of 4 stages, each consisting of 16 substages.

We divide the 512-bit block b into 32-bit words $b_1 b_2 \dots b_{16}$.

Each of the 16 substages of stage i uses one of the 32-bit words of b , but the order they are used is defined by a permutation π_i that depends on i .

In particular, substage j of stage i uses word b_ℓ , where $\ell = \pi_i(j)$ to update the state vector s .

The new state is $f_{i,j}(s, b_\ell)$, where $f_{i,j}$ is a bit-scrambling function that depends on i and j .

Further remarks on MD5

We omit further details of the bit-scrambling functions $f_{i,j}$,

However, note that the state s can be represented by four 32-bit words, so the arguments to $f_{i,j}$ occupy only 5 machine words. These easily fit into the high-speed registers of modern processors.

The definitive specification for MD5 is RFC1321 and errata. A general discussion of MD5 along with links to recent work and security issues can be found on Wikipedia.

Although MD5 is widely used, recent attacks by Xiaoyun Wang and Hongbo Yu show that it is not collision resistant and hence no longer suitable for most cryptographic uses.

Digital Signatures with Special Properties

Electronic voting

Voting involves several parties and steps:

- ▶ The election authority issues a ballot.
- ▶ The voter marks and submits the ballot.
- ▶ The election authority counts the valid ballots.

Security requirements:

1. Marked ballot should remain secret and not linked to voter.
2. All counted ballots should be genuine.

Validating ballots

To assure that only valid ballots are counted, the election authority can sign each ballot. However, there is a problem.

- ▶ If all blank ballots are identical, then they can be duplicated, allowing the voter to vote multiple times.
- ▶ If each ballot is unique (say by having a serial number), then the marked ballot can be linked to the voter, violating voter privacy.

Blind signatures

Blind signatures allow the unlinking of a message content from the signer. Here's how it works:

1. V_i takes a blank ballot, marks her choices, and then encrypts the marked ballot b with a secret random *blinding factor* r .
2. The election authority *validates* the blinded ballot b' with a signature s' .
3. V_i removes the blinding factor to get a valid signature s for b and anonymously submits (b, s) .

Since only V_i knows r , the election authority cannot link b' with b , so her vote remains secret.

Still, multiple voting is prevented since invalid ballots will not be counted, and the election authority will only sign one ballot for V_i .

RSA blind signatures

Assume Sam uses an RSA signature scheme with modulus n , private signing key d , and public verification key e .

Here's how Vi and Sam can work together to produce an unlinkable signature s for Vi's ballot b .

1. Vi generates a random $r \in \mathbf{Z}_n^*$ and computes $b' = br^e \pmod n$. She gives b' to Sam.
2. Sam signs b' , returning (b', s') to Vi.
3. Vi computes $s = s'r^{-1} \pmod n$.

Since we are using RSA signatures, $s' = (b')^d \pmod n$, so

$$s \equiv s'r^{-1} \equiv (b')^d r^{-1} \equiv b^d r^{ed} r^{-1} \equiv b^d \pmod n.$$

Hence, s is a valid RSA signature for b .

Electronic cash

Blind signatures were originally proposed for electronic cash.

As with voting, the goal was to allow for anonymous electronic cash – cash that could not be traced back to the spender.

In Chaum's scheme, a user gets an electronic coin (ρ, s) from the bank.

The user chooses ρ at random, and s is the bank's RSA signature of $h(\rho) \bmod n$, where h is a cryptographic hash function.

A coin is only valid if $V_e(h(\rho), s)$ holds.

Blind signatures prevent the bank from linking ρ with the user.

Producing a digital coin

The bank has an RSA signature scheme with public verification key (n, e)

1. The user generates a secret random number ρ and computes $m = h(\rho) \bmod n$.
2. The user blinds m with a random blinding factor r and gives $m' = mr^e \bmod n$ to the bank.
3. The bank signs m' , returning s' .
4. The user unblinds s' to obtain s , a valid signature of m .
5. The digital coin is the pair (ρ, s) .