Lecture Notes 16

1 Combining Signatures with Encryption

One often wants to encrypt messages for privacy and sign them for integrity and authenticity. Suppose Alice has a cryptosystem (E, D) and a signature system (S, V). Three possibilities come to mind for encrypting and signing a message m:

1. Alice signs the encrypted message, that is, she sends $(E(m), S(E(m)))$.
2. Alice encrypts the signed message, that is, she sends $E(m \circ S(m))$. 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 encrypts only the first component of the signed message, that is, she sends $(E(m), S(m))$.

Note that method 3 is quite problematic since signature functions make no guarantee of privacy. In particular, if (S, V) is, say, an RSA signature scheme, we can define a new signature scheme (S', V'):

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

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

Clearly, the ability to forge signatures in (S', V') implies the ability to forge signatures in (S, V), for if (m, s) is a valid signed message in (S', V'), then (m, t) is a valid signed message in (S, V), where t is the second component of the ordered pair encoded by s. Thus, the new scheme is at least as secure as the original. But with (S', V'), the plaintext message is part of the signature itself, so if (S', V') is used as the signature scheme in method 3 above, there is no privacy.

Think about the pros and cons of the other two possibilities.

2 ElGamal Signatures

The ElGamal signature scheme uses ideas similar to those of his encryption system, which we have already seen. The private signing key consists of a primitive root g of a prime p and an exponent x. The public verification key consists of g, p, and the number $a = g^x \bmod p$. The signing and verification algorithms are given below:

To sign m:
1. Choose random $y \in \mathbf{Z}^*_{\phi(p)}$. [1]
2. Compute $b = g^y \bmod p$.
3. Compute $c = (m - xb)y^{-1} \bmod \phi(p)$.
4. Output signature $s = (b, c)$.

To verify (m, s), where $s = (b, c)$:
1. Check that $a^b b^c \equiv g^m \pmod{p}$.

Why does this work? Plugging in for a and b, we see that

$$a^b b^c \equiv (g^x)^b (g^y)^c \equiv g^{xb+yc} \equiv g^m \pmod{p}$$

since $xb + yc \equiv m \pmod{\phi(p)}$.

3 Digital Signature Algorithm (DSA)

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 186–2)[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.

Now 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 $x \in \mathbf{Z}_q^*$ and compute $a = g^x \bmod p$. The parameters p, q, and g are common to the public and private keys. In addition, the private signing key contains x and the public verification key contains a.

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 $v = b$.

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,

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

It follows that

$$g^{u_1} a^{u_2} \bmod p = g^y \bmod p$$

and hence

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

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

[1] Recall that $\phi(p) = p - 1$ since p is prime.

[2] See http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf .

[3] The original standard specified that the length L of p should be a multiple of 64 lying between 512 and 1024. However, Change Notice 1 of FIPS 186–2 requires $L = 1024$.

Remarks

DSA introduces this new element of computing a number modulo p and then modulo q. Call this function $f_{p,q}(n) = (n \bmod p) \bmod q$. This is not the same as $n \bmod r$ for any number r, nor is it the same as $(n \bmod q) \bmod p$.

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	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	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?)

4 Common Hash Functions

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.

4.1 SHA–1

The revised Secure Hash Algorithm (SHA–1) is one of four algorithms described in U. S. Federal Information Processing Standard FIPS PUB 180–2 (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.

4.2 MD5

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-bit $\times$ 512-bit $\rightarrow$ 128-bit.

The MD5 hash function h is obtained as follows: First the original message is padded to length a multiple of 512. The result m is split into a sequence of 512-bit blocks $m_1, m_2, \ldots, m_k$. Finally, h is computed by chaining g on the first argument.

We look at these steps in greater detail. 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.

[4] See http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf .

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,

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

The basic block function $g(s, b)$ consists of 4 stages, each consisting of 16 substages. Recall that b is 512-bits long, so we may divide b into 32-bit words $b_1 b_2 \ldots b_{16}$. At stage i, substage j, a permutation π_i of $\{1, \ldots, 16\}$ is used to select word b_ℓ, where $\ell = \pi_i(j)$. A new state is generated by computing $f_{i,j}(s, b_\ell)$, where s is the old state and $f_{i,j}$ is a bit-scrambling function that depends on i and j. Since a state can be represented by four 32-bit words, the arguments to $f_{i,j}$ occupy only 5 machine words, which easily fit into the high-speed registers of modern processors.