YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 467b: Cryptography and Computer Security

Week 8 (rev. 2)
Professor M. J. Fischer *March 1 & 3, 2005*

Lecture Notes, Week 8

1 Digital Signatures

1.1 Definition

A *digital signature* is a string attached to a message that is used to guarantee the integrity and authenticity of the message. It is very much like the message authentication codes (MACs) discussed in lecture notes week 4. Recall that Alice can protect a message m (encrypted or not) by attaching a MAC $\xi = C_k(m)$ to the message m. The pair (m, ξ) is an *authenticated message*. To produce a MAC requires possession of the secret key k. To verify the integrity and authenticity of m, Bob, who also must know k, checks a received pair (m', ξ') by verifying that $\xi = C_k(m)$. Assuming Alice and Bob are the only parties who share k, then Bob knows that either he or Alice must have sent the message.

A digital signature can be viewed as a 2-key MAC, just as a public key cryptosystem is a 2-key version of a classical cryptosystem. The basic idea is the same. 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* $S_d : \mathcal{M} \to \mathcal{S}$, 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 $V_e(m, s)$ holds, and we say that (m, s) is *signed with* e.

The basic property of a signature scheme is that the signing function always produces valid signatures, that is,

$$V_e(m, S_d(m)) \tag{1}$$

always holds. Assuming d is Alice's private signing key, and only Alice knows d, then a valid message signed with Alice's key d identifies her with m (possibly erroneously, as we shall see).

1.2 RSA digital signature scheme

1.2.1 Commutative cryptosystems

RSA can be used for digital signatures as follows: Alice generates an RSA modulus n and key pair (e, d), where e is public and d private as usual. Let $S_d(m) = D_d(m)$, and let $V_e(m, s)$ hold iff $m = E_e(s)$. To see that (1) holds, we must verify that $m = E_e(D_d(m))$ for all messages m. This is the reverse of the condition we required for RSA to be a valid cryptosystem, viz. $D_d(E_e(m))$ for all $m \in \mathbf{Z}_m$. However, RSA satisfies both conditions since

$$D_d(E_e(m)) \equiv (m^e)^d \equiv m^{ed} \equiv (m^d)^e \equiv E_e(D_d(m)) \equiv m \pmod{n}.$$

A cryptosystem with this property that $D_d \circ E_e = E_e \circ D_d$ is said to be *commutative*, where "$\circ$" denotes functional composition. Indeed, any commutative public key cryptosystem can be used for digital signatures in exactly this same way.

[1] As 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 same mnemonic significance.

1.2.2 Signatures from non-commutative cryptosystems

We digress slightly and ask what we could do in case E_e and D_d did not commute. One possibility would be to define $S_e(m) = E_e(m)$ and $V_e(m, s)$ to hold iff $m = D_d(s)$. Now indeed every validly-signed message $(m, S_e(m))$ would verify since $E_e(D_d(m)) = m$ is the basic property of a cryptosystem. To make use of this scheme, Alice would have to keep e private and make d public. Assuming Alice generated the key pair in the first place, there is nothing preventing her from doing this. However, the resulting system might not be secure, for even if it is hard for Eve to find d from e, it might not be hard to find e from d. For RSA, it is just as hard, but then that's because RSA is completely symmetric in e and d. Other cryptosystems do not necessarily enjoy this symmetry property, e.g., ElGamal (discussed in lecture notes week 6). In that scheme, we have the equation $b = g^y \pmod p$. Finding y from b is the discrete log problem and is believed to be hard. But going the other way, finding b from y, is straightforward, so the roles of public and private key cannot be interchanged while preserving security.[2]

1.3 Security of digital signatures

For digital signatures to be of any use, they must be difficult to forge. Just as there are different notions of security of a cryptosystem, there are different notions of valid signatures being hard to forge. Below are some increasingly stringent notions of forgery-resistance:

- Resistance to forging valid signature for particular message m.
- Above, but where adversary knows a set of valid signed messages $(m_1, s_1), \ldots, (m_k, s_k)$.
- Above, but where adversary can choose a set of valid signed messages, specifying either the messages (corresponding to a chosen plaintext attack on a cryptosystem) or the signatures (corresponding to chosen ciphertext attack on a cryptosystem).
- Any of the above, but where one wishes to protect against generating any valid signed message (m', s') different from those already seen, not just for a particular predetermined m.

RSA signatures are indeed vulnerable to the latter kind of attack. It is easy for an attacker to generate valid random signed messages (m', s'). He merely chooses s' at random and computes $m' = E_e(s')$.

One often wants to sign random strings, so this is a real drawback for certain practical applications. For example, in the Diffie-Hellman key exchange protocol discussed in lecture notes week 6, Alice and Bob exchange random-looking numbers $a = g^x \bmod p$ and $b = g^y \bmod p$. In order to discourage man-in-the-middle attacks, they may wish to sign these strings. (This assumes that they already have each other's public signature verification keys.) However, Mallory could feed bogus signed values a' and b' to Bob and Alice, respectively. The signatures would check, and both would think they had successfully established a shared key k when in fact they had not.

One way to get around the adversary's ability to generate valid random signed messages is to put redundancy into the message, for example, by prefixing a fixed string σ to the front of each message before signing it. Instead of taking $S_d(m) = D_d(m)$, one could take $S_d(m) = D_d(\sigma m)$. The corresponding verification predicate $V_e(m, s)$ would then check that $\sigma m = E_e(s)$.

The security of this scheme depends on the mixing properties of the encryption and decryption functions, that is, each output bit depends on all of the input bits. Not all cryptosystems have this

[2] However, ElGamal found a different way to use the ideas of discrete logarithm to build a signature scheme, which we will discuss later.

property. For example, a block cipher used in ECB mode (see lecture notes week 2) encrypts a block at a time, so each block of output bits depends only on the corresponding block of input bits. With such a cipher, it could well happen that $S_d(m) = D_d(\sigma m) = D_d(\sigma) \cdot D_d(m)$. This would allow Mallory to forge random messages assuming he knows just one valid signed message (m_0, s_0). Here's how. He knows that $s_0 = D_d(\sigma) \cdot D_d(m)$, so from s_0 he extracts the prefix $s_{00} = D_d(\sigma)$. He now generates a valid signed message by choosing a ransom s'_{01} and computing $m' = E_e(s'_{01})$ and $s' = s_{00} \cdot s'_{01}$. The signed message (m', s') is valid since $E_e(s') = E_e(s_{00} \cdot s'_{01}) = E_e(s_{00}) \cdot E_e(s'_{01}) = \sigma m'$.

A better way to eliminate Mallory's ability to generate valid signed messages is to sign a *message digest* of the message rather than signing m itself. A message digest function h, also called a *cryptographic hash 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 $h(m) = E_e(s)$. For Mallory to generate a forged signed message (m', s') he must somehow come up with m' and s' satisfying

$$h(m') = E_e(s')$$

That is, he must find m' and s' that both map to the same string, where m' is mapped by h and s' by E_e. Alice can invert E_e by computing D_d, which is what enables her to sign messages. But Mallory presumably cannot compute D_d. The defining property of a cryptographic hash function implies that he also can't "invert" h. h is not one-to-one, so it doesn't have an inverse in the usual sense. What we mean here is that, given y, Mallory cannot feasibly find any m' such that $h(m') = y$.

A second advantage of signing message digests rather than signing messages directly is that the resulting signed messages are shorter. If one uses RSA in the way described to sign m, the signature is same length as m, so the signed message is twice as long as m. But a message digest is a fixed length, for example, 160 bits, so the signed version of m is only 160 bits longer than m itself. For both reasons of security and efficiency, signed message digests are what is used in practice.

We'll talk more about message digests in section 3.

2 Implications of Digital Signatures

We like to think of a digital signature as a digital analog to a conventional signature. However, there is an important difference. A conventional signature binds a person to a document. Barring forgery, a valid signature indicates that a particular individual performed the action of signing the document. Similarly, a digital signature binds a signing key to a document. Barring forgery, a valid digital signature indicates that a particular signing key was used to sign the document. However, it does *not* directly bind an individual to the document the way a conventional signature does. Rather, the binding of an individual to a document is indirect. The signature binds the signing key to the document, but other considerations must be used to bind the individual to the signing key.

An individual can always disavow a signature on the grounds that the private signing key has become compromised. There are many ways that this can happen. Since signing keys must be quite long for security (768, 1024, or even 2048 bits are common), Alice can't be expected to memorize the key; it must be written down somewhere. Whereever it resides, there is the possibility of it being copied, or the physical device containing it being stolen. Even if she were able to memorize it, she would have to type it into her computer in order to use it, exposing it to keystroke monitors or other spyware that might have been surreptitiously installed on her computer. Indeed, she might deliberately publish her signing key for the express purpose of relinquishing responsibility for documents signed by her key. For all of these reasons, one cannot conclude "without a reasonable

doubt" that a digitally signed document was indeed signed by the purported holder of the signing key.

This isn't to say that digital signatures aren't useful; only that they have significantly different properties than conventional signatures. In particular, they are subject to disavowal by the signer in a way that conventional signatures are not. Nevertheless, they are still very useful in situations where disavowal is not a problem.

3 Message Digest Functions

Recall that a *message digest* or *cryptographic one-way hash* function h maps $\mathcal{M} \to \mathcal{H}$, where $\mathcal{M}$ is the message space and $\mathcal{H}$ the hash value space. A *collision* is a pair of messages m_1 and m_2 such that $h(m_1) = h(m_2)$, and we say that m_1 and m_2 *collide*. We generally assume $|\mathcal{M}| \gg |\mathcal{H}|$, in which case h is very far from being one-to-one, and there are many colliding pairs. Nevertheless, we want it to be hard for an adversary to find collisions. We consider three increasingly strong versions of this property:

One-way property: Given $y \in \mathcal{H}$, it is "hard" to "find" $m \in \mathcal{M}$ such that $h(m) = y$.

Weak collision-free property: Given $m \in \mathcal{M}$, it is "hard" to "find" $m' \in \mathcal{M}$ such that $m' \neq m$ and $h(m') = h(m)$.

Strong collision-free property: It is "hard" to "find" colliding pairs (m, m'). (What does this mean in a complexity-theoretic sense?)

These definitions as I have stated them are rather vague, for they ignore issues of what we mean by "hard" and "find". Intuitively, "hard" means that Mallory cannot carry out the computation in a feasible amount of time on a realistic computer, but this just raises the question with what do we mean by "feasible time" and "realistic computer"? The term "find" has several reasonable interpretations. It may mean "always produces a correct answer", or "produces a correct answer with high probability", or "produces a correct answer on a significant number of possible inputs with non-negligible probability". The latter notion of "find" is rather weak; it just says that Mallory every now and then can break the system. For any given application, there is a maximum acceptable rate of error, and we must be sure that our cryptographic system meets that requirement.

Let me say a little more about what it means for h to be one-way. Intuitively, this means that no probabilistic polynomial time algorithm $A(y)$ produces a pre-image m of y under h with more than negligable probability of success. However, this probability of success need not be negligable for all y. We only require that the expected probability of success be negligable when y itself is chosen according to a particular probability distribution. The distribution we have in mind is the one induced by h applied to uniformly distributed $m \in \mathcal{M}$. That is, the probability of y is proportional to the number of values $m \in \mathcal{M}$ such that $h(m) = y$. This means that h can be considered one-way even though algorithms do exist that succeed on low-probability subsets of $\mathcal{H}$.

The following example might help clarify these ideas. Let $h(m)$ be a cryptographic hash function that produces hash values of length n. Define a new hash function $H(m)$ as follows:

$$H(m) = \begin{cases} 0 \cdot m & \text{if } |m| = n \\ 1 \cdot h(m) & \text{otherwise.} \end{cases}$$

Thus, H produces hash values of length $n+1$. $H(m)$ is clearly collision-free since the only possible collisions are for m's of lengths different from n. Any such colliding pair (m, m') for H is also a colliding pair for h. Since h is collision-free, then so is H.

Not so obvious is that H is one-way, *even though H can be inverted for 1/2 of all possible hash values y*, namely, those which begin with 0. The reason this doesn't violate the definition of one-wayness is that only 2^n values of m map to hash values that begin with 0, and all the rest map to values that begin with 1. Since we are assuming $|\mathcal{M}| \gg |\mathcal{H}|$, the probability that a uniformly sampled $m \in \mathcal{M}$ has length exactly n is small.

With these caveats, there are some obvious relationships between these definitions that can be made precise once the definitions are made similarly precise.

Claim 1 *If h is strong collision-free, then h is weak collision-free.*

Proof: (Sketch) Suppose h is *not* weak collision-free. We show that it is not strong collision-free by showing how to enumerate colliding message pairs. The method is straightforward: Pick a random message $m \in \mathcal{M}$. Try to find a colliding message m'. If we succeed in finding such an m', then output the colliding pair (m, m'). If not, try again with another randomly-chosen message. Since h is not weak collision-free, we will succeed on a significant number of the messages, so we will succeed in generating a succession of colliding pairs. How fast the pairs are enumerated depends on how often the algorithm for finding a colliding message succeeds and how fast it is.

These parameters in turn may depend on how large $\mathcal{M}$ is relative to $\mathcal{H}$. It is always possible that h is one-to-one on some subset U of elements in $\mathcal{M}$, so not every message even necessarily has a colliding partner. However, an easy counting argument shows that U has size at most $|\mathcal{H}| - 1$. Since we assume $|\mathcal{M}| \gg |\mathcal{H}|$, the probability that a randomly-chosen message from $\mathcal{M}$ lies in U is correspondingly small. ■

In a similar vein, we argue that strong collision-free also implies one-way.

Claim 2 *If h is strong collision-free, then h is one-way.*

Proof: (Sketch) Suppose h is *not* one-way. Then there is an algorithm $A(y)$ for finding m such that $h(m) = y$, and $A(y)$ succeeds with significant probability when y is chosen randomly with probability proportional to the size of its preimage. Assume that $A(y)$ returns $\perp$ to indicate failure.

The following randomized algorithm enumerates a sequence of colliding pairs:

1. Choose random m.
2. Compute $y = h(m)$.
3. Compute $m' = A(y)$.
4. If $m' \neq \perp$ and $m \neq m'$ then output (m, m').
5. Start over at step 1.

Each iteration of this algorithm succeeds with significant probability ε that is the product of the probability that $A(y)$ succeeds on y and the probability that $m' \neq m$. The latter probability is at least 1/2 except for those values m which lie in the set of U of messages on which h is one-to-one (defined in the proof of claim 1). Thus, assuming $|\mathcal{M}| \gg |\mathcal{H}|$, the algorithm outputs each colliding pair in expected number of iterations that is at most only slightly larger than $1/\varepsilon$. ■

These same ideas can be used to show that weak collision-free implies one-way, but now one has to be more careful with the precise definitions.

Claim 3 *If h is weak collision-free, then h is one-way.*

Proof: (Sketch) Suppose as before that h is *not* one-way, so there is an algorithm $A(y)$ for finding m such that $h(m) = y$, and $A(y)$ succeeds with significant probability when y is chosen randomly with probability proportional to the size of its preimage. Assume that $A(y)$ returns $\perp$ to indicate failure. We want to show this implies that the weak collision-free property does not hold, that is, there is an algorithm that, for a significant number of $m \in \mathcal{M}$, succeeds with non-negligible probability in finding a colliding m'. We claim the following algorithm works:

Given input m:
1. Compute $y = h(m)$.
2. Compute $m' = A(y)$.
3. If $m' \neq \perp$ and $m \neq m'$ then output (m, m') and halt.
4. Otherwise, start over at step 1.

This algorithm fails to halt for $m \in U$, but we have argued above that the number of such m is small (= insignificant) when $|\mathcal{M}| \gg |\mathcal{H}|$. It may also fail even when a colliding partner m' exists if it happens that the value returned by $A(y)$ is m. (Remember, $A(y)$ is only required to return some preimage of y; we can't say which.) However, corresponding to each such bad case is another one in which the input to the algorithm is m' instead of m. In this latter case, the algorithm succeeds, since y is the same in both cases. With this idea, we can show that the algorithm succeeds in finding a colliding partner on at least half of the messages in $\mathcal{M} - U$. ■

4 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 $(E(m), S(m))$.

Think about the pros and cons of each of these three possibilities.

5 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 $y \in \mathbf{Z}^*_{\phi(p)}$. [3]
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 $xb + yc \equiv m \pmod{\phi(p)}$.

6 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). (See `http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf`.) It uses two primes: p, which is 1024 bits long,[4] 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 qth 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.)

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

[4] 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?)