CPSC 467b: Cryptography and Computer Security
Lecture 7

Michael J. Fischer

Department of Computer Science
Yale University

February 1, 2010

Message authentication codes (MACs)

What Alice and Bob need to solve their problem is called a *Message Authentication Code* or *MAC*.

A MAC is generated by a function $C_k(m)$ that can be computed by anyone knowing the secret key k.

It should be hard for an attacker, without knowing k, to find any pair (m, ξ) such that $\xi = C_k(m)$.

This should remain true even if the attacker knows a set of valid MAC pairs $\{(m_1, \xi_1), \ldots, (m_t, \xi_t)\}$ so long as m itself is not the message in one of the known pairs.

Computing a MAC

A block cipher such as DES can be used to compute a MAC by making use of one of the ciphertext chaining modes CBC or CFB.

In these modes, the last ciphertext block c_t depends on all t message blocks $m_1, \ldots, m_t$, so, we define $C_k(m) = c_t$. The function so computed is reputed to be a good MAC.

Note that the MAC is only a single block long. This is in general much shorter than the message. A MAC acts like a checksum for preserving data integrity, but it has the advantage that an adversary cannot compute a valid MAC for an altered message.

Using a MAC

Sending an unencrypted authenticated message.

- Alice sends m in the clear along with the MAC $\xi = C_k(m)$.
- Bob receives m' and ξ', possibly different from m and ξ.
- Bob checks that $\xi' = C_k(m')$ and if so, accepts m' as a valid message from Alice.

Mallory *successfully cheats* if Bob accepts a message m' as valid that Alice never sent.

The assumed property of a MAC is that Mallory cannot do this, even knowing a set of valid MAC pairs previously sent by Alice.

The MAC prevents forgery of messages but does not protect privacy.

Protecting both privacy and authenticity

If Alice wants both privacy and authenticity, she can encrypt m and use the MAC to protect the ciphertext from alteration.

- Alice sends $c = E_k(m)$ and $\xi = C_k(c)$.
- Bob, after receiving c' and ξ', only decrypts c' after first verifying that $\xi' = C_k(c')$.

A flawed use of a MAC

Another possibility is for Alice to send $c = E_k(m)$ and $\xi = C_k(m)$. Here, the MAC is computed from m, not c.

Bob, upon receiving c' and ξ', first decrypts c' to get m' and then checks that $\xi' = C_k(m')$, i.e., Bob checks $\xi' = C_k(D_k(c'))$

In practice, this might also work, but its security does *not* follow from the assumed security property of the MAC.

The MAC property says Mallory cannot produce a pair (m', ξ') for an m' that Alice never sent. It does *not* follow that he cannot produce a pair (c', ξ') that Bob will accept as valid, even though c' is not the encryption of one of Alice's messages.

If Mallory succeeds in convincing Bob to accept (c', ξ'), then Bob will decrypt c' to get $m' = D_k(c')$ and incorrectly accept m' as coming from Alice.

Example of a flawed use of a MAC

Here's how Mallory might find (c', ξ') such that $\xi' = C_k(D_k(c'))$.

Suppose the MAC function C_k and encryption function E_k are the same. Then $C_k(D_k(c')) = E_k(D_k(c')) = c'$, so Bob accepts every pair (c', c') as valid![1]

A more plausible example is where C_k is derived from E_k using the CBC or CFB chaining modes as described earlier. In that case, the MAC is the last ciphertext block c'_t, and Bob will always accept (c', c'_t) as valid.

[1] The astute reader will notice that $E_k(D_k(c))$ might differ from c for c not in the range of E_k. However, in most of the cryptosystems we consider, the message space $\mathcal{M}$ and ciphertext space $\mathcal{C}$ are the same. When that is the case, the range of E_k is all of $\mathcal{C}$, so every $c \in \mathcal{C}$ can be written as $c = E_k(y)$ for some message y, and $E_k(D_k(c)) = E_k(D_k(E_k(y))) = E_k(y) = c$.

Asymmetric cryptosystems

An asymmetric cryptosystem has a pair $k = (k_e, k_d)$ of related keys, the *encryption key k_e* and the *decryption key k_d*.

Alice encrypts a message m by computing $c = E_{k_e}(m)$.
Bob decrypts c by computing $m = D_{k_d}(c)$.

- We sometimes write e instead of k_e and d instead of k_d, e.g., $E_e(m)$ and $D_d(c)$.
- We sometimes write k instead of k_e or k_d where the meaning is clear from context, e.g., $E_k(m)$ and $D_k(c)$.

In practice, it isn't generally as confusing as all this, but the potential for misunderstanding is there.

As always, the decryption function inverts the encryption function, so $m = D_d(E_e(m))$.

Security requirement

Should be hard for Eve to find m given $c = E_e(m)$ *and e*.

- The system remains secure even if the encryption key e is made public!
- Asymmetric cryptosystems are sometimes called *2-key* or *public key* cryptosystems. k_e is the *public key* and k_d the *private key*.

Reason to make e public.

- Anybody can send a private message to Bob. Sandra obtains Bob's public key e and sends $c = E_e(m)$ to Bob.
- Bob recovers m by computing $D_d(c)$, using his private key d.

This greatly simplifies key management. No longer need a secure channel between Alice and Bob for the initial key distribution (which I have carefully avoided talking about so far).

Man-in-the-middle attack against 2-key cryptosystem

An active adversary Mallory can carry out a nasty *man-in-the-middle* attack.

- Mallory sends his own encryption key to Sandra when she attempts to obtain Bob's key.
- Not knowing she has been duped, Sandra encrypts her private data using Mallory's public key, so Mallory can read it (but not Bob)!
- To keep from being discovered, Mallory intercepts each message from Sandra to Bob, decrypts using his own decryption key, re-encrypts using Bob's public encryption key, and sends it on to Bob. Bob, receiving a validly encrypted message, is none the wiser.

Passive attacks against a 2-key cryptosystem

Making the encryption key public also helps a passive attacker.

1. Chosen-plaintext attacks are available since Eve can generate as many plaintext-ciphertext pairs as she wishes using the public encryption function $E_e()$.
2. The public encryption function also gives Eve a foolproof way to check the validity of a potential decryption. Namely, Eve can verify $D_d(c) = m_0$ for some candidate message m_0 by checking that $c = E_e(m_0)$.
   Redundancy in the set of meaningful messages is no longer necessary.

Facts about asymmetric cryptosystem

Good asymmetric cryptosystems are much harder to design than good symmetric cryptosystems.

All known asymmetric systems are orders of magnitude slower than corresponding symmetric systems.

Hybrid cryptosystems

Asymmetric cryptosystems are often used in conjunction with a symmetric cipher to form a *hybrid* system. Let (E^2, D^2) be a 2-key cryptosystem and (E^1, D^1) be a 1-key cryptosystem.

Here's how Alice sends a secret message m to Bob.

- Alice generates a random *session key* k for use with the 1-key system.
- Alice computes $c_1 = E^1_k(m)$ and $c_2 = E^2_e(k)$, where e is Bob's public key, and sends (c_1, c_2) to Bob.
- Bob computes $k = D^2_d(c_2)$ using his private decryption key d and then computes $m = D^1_k(c_1)$.

This is much more efficient than simply sending $E^2_e(m)$ when m is much longer than k.

Note that the 2-key system is used to encrypt random messages!

Overview of RSA

Probably the most commonly used asymmetric cryptosystem today is *RSA*, named from the initials of its three inventors, Rivest, Shamir, and Adelman.

Unlike the symmetric systems we have been talking about so far, RSA is based not on substitution and transposition but on arithmetic involving very large integers — numbers that are hundreds or even thousands of bits long.

To understand how RSA works requires knowing a bit of number theory, which I will be presenting in the next few lectures. However, the basic ideas can be presented quite simply, which I will do now.

RSA spaces

The message space, ciphertext space, and key space for RSA is the set of integers between 0 and $n - 1$ for some very large integer n.

For now, think of n as a number so large that its binary representation is 1024 bits long.

Encoding bit strings by integers

To use RSA as a block cipher to send bit strings, Alice must convert each block to an integer m, and Bob must convert m back to a block.

Many such encodings are possible, but perhaps the simplest is to prepend a “1” to the block x and regard the result as a binary integer m.

To decode m to a block, write out m in binary and then delete the initial “1” bit.

To ensure that $m < n$ as required, we limit the length of our blocks to 1022 bits.

RSA key generation

Here's how Bob generates an RSA key pair.

- Bob chooses two sufficiently large distinct prime numbers p and q and computes $n = pq$.
  For security, p and q should be about the same length (when written in binary).
- He computes two numbers e and d with a certain number-theoretic relationship.
- The public key is the pair $k_e = (e, n)$. The private key is the pair $k_d = (d, n)$. The primes p and q are no longer needed and should be discarded.

RSA encryption and decryption

To encrypt, Alice computes $c = m^e \bmod n$.[2]

To decrypt, Bob computes $m = c^d \bmod n$.

This works because, for all m, we have

$$m = (m^e \bmod n)^d \bmod n. \tag{1}$$

Here, $a \bmod n$ denotes the remainder when a is divided by n.

That's all there is to it, once the keys have been found.
Most of the complexity in implementing RSA has to do with key generation, which fortunately is done only infrequently.

[2]In the remainder of this discussion, messages and ciphertexts will refer to integers in the range 0 to $n - 1$, not to bit strings.

RSA questions

You should already be asking yourself the following questions:

- How does one find n, e, d such that equation 1 is satisfied?
- Why is RSA believed to be secure?
- How can one implement RSA on a computer when most computers only support arithmetic on 32-bit or 64-bit integers, and how long does it take?
- How can one possibly compute $m^e \bmod n$ for 1024 bit numbers. m^e, before taking the remainder, has size roughly

$$\left(2^{1024}\right)^{2^{1024}} = 2^{1024\times 2^{1024}} = 2^{2^{10}\times 2^{1024}} = 2^{2^{1034}}.$$

This is a number that is roughly 2^{1034} bits long! No computer has enough memory to store that number, and no computer is fast enough to compute it.

Tools needed to answer RSA questions

Two kinds of tools are needed to understand and implement RSA.

Algorithms: Need clever algorithms for primality testing, fast exponentiation, and modular inverse computation.

Number theory: Need some theory of Z_n, the integers modulo n, and some special properties of numbers n that are the product of two primes.