YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

CPSC 467b: Cryptography and Computer Security

Week 4 (rev. 4)

Professor M. J. Fischer *February 1 & 3, 2005*

Lecture Notes, Week 4

1 Message authentication codes (MACs)

So far in the course, we have been discussing only a single cryptographic application, namely, secret message transmission from Alice to Bob over a a publicly-readable channel. Our goal has been to maintain privacy in the face of an eavesdropper Eve.

Once we assume that Eve has the power to modify messages and generate her own messages as well as eavesdrop, life becomes more difficult. We usually call that kind of a malicious adversary "Mallory" (to distinguish him from Eve, who only eavesdrops).

Encryption alone no longer solves Alice and Bob's problem. Alice sends $c = E_k(m)$, but Bob may receive a corrupted or forged $c' \neq c$. How then does Bob know that the message he receives really was sent by Alice?

The naive answer is that Bob computes $m' = D_k(c')$, and if m' "looks like" a valid message, then Bob accepts it as having come from Alice. The reasoning here is that Mallory, not knowing k, could not possibly have produced a valid-looking message.

For any particular cipher such as DES, that assumption may or may not be valid, but here are two things to watch out for:

1. There are three successively easier possible attacks in which Mallory might produce fraudulent messages:

   (a) He might produce $c' = E_k(m')$ for a message m' of his choosing.

   (b) He might produce a message c' for which the corresponding plaintext m' is a valid message, even though he could not choose m' in advance, nor perhaps he does not even know what m' is.

   (c) He might be able to alter a legitimate message c from Alice to produce a new message c' that corresponds to an altered form m' of the true message m. For example, if m represents an amount of money, it is conceivable that Mallory could find the encryption of $m + 1$ given the encryption of m, without knowing either m or $m + 1$.

   Attack (1a) is similar to, but not the same as, the notion of breaking the cryptosystem that we have been studying. We have been asking that it be hard for Eve to compute $m = D_k(c)$ given c. To carry out attack (1a) requires that Mallory compute $E_k(m)$ given m. It's conceivable that he could do the latter without being able to do the former.

   One form of attack (1b) is clearly possible, the so-called a *replay* attack. This is when Mallory substitutes a legitimate old encrypted message c' for the current message c. It can be thwarted by adding timestamps and/or sequence numbers to the messages, so that Bob can recognize when old messages are being received. Of course, this only works if Alice and Bob anticipate the attack and incorporate appropriate countermeasures into the protocol they are using.

   However, even if replay attacks are ruled out, a cryptosystem that is secure against attack (1a) might still permit attack (1b). There are all sorts of ways that Mallory can generate values c'. What gives us confidence that Bob won't accept one of them as being valid?

Attack (1c) might be possible even in a cryptosystem that is free from attacks (1a) and (1b). For example, if c_1 and c_2 are encryptions of valid messages, perhaps so is $c_1 \oplus c_2$. Whether or not it is depends entirely on particular properties of E_k. It does not follow in general from the difficulty of decrypting a given ciphertext. We will see some cryptosystems later which do have the property of being vulnerable to attack (1c). In some contexts, this can actually be a useful property, as we will see.

2. Cryptosystems are not always used to send natural language or other highly-redundant messages. For example, suppose Alice wants to send Bob her password to a web site. Knowing full well the dangers of sending passwords in the clear over the internet, she chooses to encrypt it instead. Since passwords are supposed to look like random strings of characters, Bob will likely accept anything he gets from Alice. He could be quite embarrassed (or worse) claiming he knew the correct password when in fact the password he thought was from Alice was actually a fraudulent one derived from a random ciphertext c' produced by Mallory.

What Alice and Bob need to solve their problem is called a *Message Authentication Code* or *MAC*. A MAC is generated by a function function $C_k(m)$ that can be computed by anyone knowing a secret key k. However, it should be hard for an attacker to find any pair (m, ξ) such that $\xi = C_k(m)$ without knowing k. In fact, this should remain hard 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. We will discuss next time how DES and other block ciphers can be used to generate a MAC.

Using a MAC, Alice can send both $c = E_k(m)$ and also $\xi = C_k(m)$. Bob receives c' and ξ', possibly different from what Alice sent. Bob computes $m' = D_k(c')$ and then checks that $\xi' = C_k(m')$. If the check fails, then Bob knows that either m or ξ (or both) are invalid.

2 Computing MACs

One way to use a block cipher to compute a MAC is to use one of the ciphertext chaining modes, CBC or CFB. (See Lecture notes week 2, section 2.2.) In these modes, the last ciphertext block c_t depends on all t message blocks $m_1, \ldots, m_t$. Therefore, we define $C_k(m) = c_t$. The result of this process is reputed to be a good MAC generation function. Note that the MAC is only a single block long, which in general is 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.

3 Asymmetric cryptosystems

A major advance in cryptography is the modern development of *asymmetric* (also called *2-key* or *public key*) cryptosystems. The idea is simple. Instead of having a single key k that is used by both Alice and Bob, an asymmetric cryptosystem has a pair of related keys, an *encryption key* e and a *decryption key* d. Alice encrypts a message m by computing $c = E_e(m)$. Bob decrypts by computing $m = D_d(c)$.[1] As always, the decryption function inverts the encryption function, so the

[1] We often get sloppy with notation when discussing asymmetric cryptosystems. Let $k = (e, d)$ be a key pair. We sometimes write k_e and k_d for the first and second keys, respectively, so we might use the rather cumbersome notation $E_{k_e}(m)$ and $D_{k_d}(c)$. But then we might simplify this by dropping the second-level subscripts to get the same notation we use for symmetric cryptosystems, namely $E_k(m)$ and $D_k(c)$. Nevertheless, it should still be understood that the "k" in E_k refers to the first element of the key pair, whereas the "k" in D_k refers to the second. In practice, it isn't generally as confusing as all this. but the potential for misunderstanding is there.

following is always satisfied:

$$m = D_d(E_e(m)).$$

What makes asymmetric systems useful is the additional requirement that it be difficult to find d from e, and more generally, that it be difficult to find m given both e and $c = E_e(m)$. Thus, the system remains secure even if the encryption key e is made public!

There are several reasons to make e public. One is that it is then possible for anybody to send a private message to Bob. Sandra need only obtain Bob's public key e and send Bob $E_e(m)$. Bob recovers m by applying D_d to the received ciphertext. This greatly simplifies the key management problem, for there is no longer the need for a secure channel between Alice and Bob for the initial key distribution (which I have carefully avoided talking about so far).

Of course, an active adversary Mallory can still do some nasty things. For example, he could send his own encryption key to Sandra when she attempts to obtain Bob's key. Not knowing she has been duped, Sandra would then encrypt her private data in a way that Bob could not read but Mallory could! If Mallory wanted to carry out this charade for a longer time without being discovered, he could intercept each message from Sandra to Mallory, decrypt the message using his own decryption key, then re-encrypt it using Bob's public encryption key and send it on its way to Bob. Bob, receiving a validly encrypted message, will be none the wiser to Mallory's shenanigans. This is an example of a *man-in-the-middle* attack.

The security requirements for an asymmetric cryptosystem are more stringent than for a symmetric cryptosystem. For example, the system must be secure against large-scale chosen plaintext attacks, for Eve can generate as many plaintext-ciphertext pairs as she wishes using the public encryption function $E_e()$.

The public encryption function also gives Eve a way to check the validity of a potential decryption. Namely, if Eve guesses that $D_d(c) = m_0$ for some candidate message m_0, she can check her guess by testing if $c = E_e(m_0)$. Thus, whether or not there is redundancy in the set of meaningful messages is of no consequence to her since she now has an independent way of determining a successful decryption.

Designing a good asymmetric cryptosystem is much harder than designing a good symmetric cryptosystem. All of the known asymmetric systems are orders of magnitude slower than corresponding symmetric systems. Therefore, they are often used in conjunction with a symmetric cipher to form a *hybrid* system. Here's how this works. Suppose (E^2, D^2) is a 2-key cryptosystem and (E^1, D^1) is a 1-key cryptosystem. To send a secret message m to Bob, Alice first generates a random *session key* k for use with the 1-key system. which she then uses to encrypt m. She then encrypts the session key using Bob's public key for the 2-key system and sends Bob both ciphertexts. In formulas, she sends Bob $c_1 = E^1_k(m)$ and $c_2 = E^2_e(k)$. Bob decrypts c_2 using $D^2_d()$ to obtain k and then decrypts c_1 using $D^1_k()$ to obtain m. This is much more efficient than simply sending $D^2_e(m)$ in the common case that m is much longer than k.

4 RSA

Probably the most commonly used asymmetric cryptosystem in use 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 we will be presenting in the next few lectures. However, the basic ideas can be presented quite simply, which I will do now.

RSA assumes the message space, ciphertext space, and key space are the set of integers between 0 and $n-1$, where n is a large integer. For now, think of n as a number so large that its binary representation is 1024 bits long. To use RSA in the usual case where we are interested in sending bit strings, Alice must first convert her message to an integer, and Bob must convert the integer he gets from decryption back to a bit string. Many such encodings are possible, but perhaps the simplest is to prepend a "1" to the bit string x and regard the result as a binary integer m. To decode m to a bit string, write out m in binary and then delete the initial "1" bit. To ensure that $m < n$ as required, we will limit the length of our binary messages to 1022 bits.

Here's how RSA works. Alice chooses two sufficiently large prime numbers p and q and computes $n = pq$. For security, p and q should be about the same length (when written in binary). She then computes two numbers e and d with a certain number-theoretic relationship. The public key is the pair (e, n) The private key is the pair (d, n). The primes p and q are no longer needed and should be discarded.

To encrypt, Alice computes $c = m^e \bmod n$.[2] To decrypt, Bob computes $m = c^d \pmod n$. Here, $a \bmod n$ means the remainder when a is divided by n. That's all there is to it, once the keys have been found. It turns out that most of the complexity in implementing RSA has to do with key generation, which fortunately is done only infrequently.

You should already be asking yourself the following questions:

- How does one find p, q, e, d with the desired properties?
- What are the desired properties that make RSA work? A priori, it seems pretty unlikely that $D_d(E_e(m)) = (m^e \bmod n)^d \bmod n = m$.
- Why is RSA believed to be secure?
- How can one implement RSA on a computer when most computers only support arithmetic on 32-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, is a number that is roughly 2^{1024} bits long. No computer has enough memory to store that number, and no computer is fast enough to compute it.

To answer these questions will require the study of clever algorithms for primality testing, fast exponentiation, and modular inverse computation. It will also require some theory of Z_n, the integers modulo n, and some properties of numbers n that are the product of two primes. In particular, the security of RSA is based on the premise that the *factoring problem* on large integers is infeasible, that is, given n that is known to be the the product of two primes p and q, to find p and q.

5 Number Theory Review

We next review some number theory that is needed for understanding RSA. These lecture notes only provide a high-level overview. Further details are contained in course handouts 3–5 and in Chapters 5 and 6 of the textbook.

[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.

5.1 Basic definitions

$$\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$$

$\mathbf{Z}_n$ is an Abelian group under addition (+).

$$\mathbf{Z}_n^* = \{x \in \mathbf{Z}_n \mid \gcd(x, n) = 1\}$$

$\mathbf{Z}_n^*$ is an Abelian group under multiplication ($\cdot$). Euler's totient (ϕ) function is defined to be the cardinality of $\mathbf{Z}_n^*$:

$$\phi(n) = |\mathbf{Z}_n^*|$$

Properties of $\phi(n)$:

1. If p is prime, then $\phi(p) = p - 1$.
2. More generally, if p is prime and $k \geq 1$, then $\phi(p^k) = p^k - p^{k-1} = (p-1)p^{k-1}$.
3. If $\gcd(m, n) = 1$, then $\phi(mn) = \phi(m)\phi(n)$.

These properties enable one to compute $\phi(n)$ for all $n \geq 1$ provided one knows the factorization of n. For example,

$$\phi(126) = \phi(2)\phi(3^2)\phi(7) = (2-1)(3-1)(3^{2-1})(7-1) = 1 \cdot 2 \cdot 3 \cdot 6 = 36.$$

The 36 elements of $\mathbf{Z}_{126}^*$ are: 1, 5, 11, 13, 17, 19, 23, 25, 29, 31, 37, 41, 43, 47, 53, 55, 59, 61, 65, 67, 71, 73, 79, 83, 85, 89, 95, 97, 101, 103, 107, 109, 113, 115, 121, 125.

5.2 Modular arithmetic

There are several closely-related notions associated with "mod".

First of all, mod is a binary operator. If $a \geq 0$ and $b \geq 1$ are integers, then $a \bmod b$ is the remainder of a divided by b. When either a or b is negative, there is no consensus on the definition of mod. We are only interested in mod for positive b, and we find it convenient in that case to define $(a \bmod b)$ to be the smallest non-negative integer r such that $a = bq + r$ for some integer q. Under this definition, we always have that $r = (a \bmod b) \in \mathbf{Z}_b$. For example $(-5 \bmod 3) = 1 \in \mathbf{Z}_3$ since for $q = -2$, we have $-5 = 3 \cdot (-2) + 1$. Note that in the C programming language, the mod operator `%` is defined differently, so $a \,\%\, b \neq a \bmod b$ when a is negative and b positive.[3]

Mod is also used to define a relationship on integers:

$$a \equiv b \pmod{n} \quad \text{iff} \quad n \,|\, a - b.$$

That is, a and b have the same remainder when divided by n. An immediate consequence of this definition is that

$$a \equiv b \pmod{n} \quad \text{iff} \quad (a \bmod n) = (b \bmod n).$$

Thus, the two notions of mod aren't so different after all!

When n is fixed, the resulting two-place relationship $\equiv$ is an equivalence relation. Its equivalence classes are called *residue* classes modulo n and are denoted using the square-bracket notation

[3] For those of you who are interested, the C standard defines $a \,\%\, b$ to be the number satisfying the equation $(a/b) * b + (a \,\%\, b) = a$. C also defines a/b to be the result of rounding the real number a/b towards zero, so $-5/3 = -1$. Hence, $-5 \,\%\, 3 = -5 - (-5/3) * 3 = -5 + 3 = -2$.

$[b] = \{a \mid a \equiv b \pmod{n}\}$. For example, for $n = 7$, we have $[10] = \{\ldots - 11, -4, 3, 10, 17, \ldots\}$. Clearly, $[a] = [b]$ iff $a \equiv b \pmod{n}$. Thus, $[-11]$, $[-4]$, $[3]$, $[10]$, $[17]$ are all names for the same equivalence class. We choose the unique integer in the class that is also in $\mathbf{Z}_n$ to be the *canonical* or preferred name for the class. Thus, the canonical name for the class containing 10 is $[10 \bmod 7] = [3]$.

The relation $\equiv \pmod{n}$ is a *congruence* relation with respect to addition, subtraction, and multiplication of integers. This means that for each of these arithmetic operations $\odot$, if $a \equiv a' \pmod{n}$ and $b \equiv b' \pmod{n}$, then $a \odot b \equiv a' \odot b' \pmod{n}$. This implies that the class containing the result of $a + b$, $a - b$, or $a \times b$ depends only on the classes to which a and b belong and not the particular representatives chosen. Hence, we can define new addition, subtraction, and multiplication as operations on equivalence classes, or alternatively, regard them as operations directly on $\mathbf{Z}_n$ defined by

$$\begin{aligned} a \oplus b &= (a + b) \bmod n \\ a \ominus b &= (a - b) \bmod n \\ a \otimes b &= (a \times b) \bmod n \end{aligned} \tag{1}$$