

CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 7
February 5, 2013

Stream cipher from block cipher

Review of OFB and CFB chaining modes

Extending chaining modes to bytes

Active adversary attacks

Steganography

Public-key cryptography

RSA

Some number theory

Factoring Assumption

Computing with Big Numbers

Stream cipher from block cipher

Output Feedback Mode (OFB)

Similar to a one-time pad, but keystream is generated from the previous block keys using E_k .

- ▶ To encrypt, Alice computes a stream of *block keys* k_1, k_2, \dots , where $k_i = E_k(k_{i-1})$ and k_0 is a fixed *initial vector* (IV).

The block keys are XORed with successive plaintext blocks.

That is, $c_i = m_i \oplus k_i$.

- ▶ To decrypt, Bob applies exactly the same method to the ciphertext to get the plaintext.

That is, $m_i = c_i \oplus k_i$, where $k_i = E_k(k_{i-1})$ and $k_0 = IV$.

Cipher-Feedback Mode (CFB)

Similar to OFB, but keystream is generated from the previous cipher text blocks using E_k .

- ▶ To encrypt, Alice computes a stream of *block keys* k_1, k_2, \dots , where $k_i = E_k(c_{i-1})$ and c_0 is a fixed *initial vector (IV)*.

The block keys are XORed with successive plaintext blocks, just as in OFB.

That is, $c_i = m_i \oplus k_i$.

- ▶ To decrypt, Bob applies exactly the same method to the ciphertext to get the plaintext.

That is, $c_i = m_i \oplus k_i$, where $k_i = E_k(c_{i-1})$ and $c_0 = IV$.

Stream ciphers from OFB and CFB block ciphers

OFB and **CFB** block modes can be turned into stream ciphers.

Both compute $c_i = m_i \oplus k_i$, where

- ▶ $k_i = E_k(k_{i-1})$ (for OFB);
- ▶ $k_i = E_k(c_{i-1})$ (for CFB).

Assume a block size of b bytes numbered $0, \dots, b-1$.

Then $c_{i,j} = m_{i,j} \oplus k_{i,j}$, so each output byte $c_{i,j}$ can be computed before knowing $m_{i,j'}$ for $j' > j$; no need to wait for all of m_i .

One must keep track of j . When $j = b$, the current block is finished, i must be incremented, j must be reset to 0, and k_{i+1} must be computed.

Extended OFB and CFB modes

Simpler (for hardware implementation) and more uniform stream ciphers result by also computing k_i a byte at a time.

The idea: Use a shift register X to accumulate the feedback bits from previous stages of encryption so that the full-sized blocks needed by the block chaining method are available.

X is initialized to some public initialization vector.

Some notation

Assume block size $b = 16$ bytes.

Define two operations: L and R on blocks:

- ▶ $L(x)$ is the leftmost byte of x ;
- ▶ $R(x)$ is the rightmost $b - 1$ bytes of x .

Extended OFB and CFB similarities

The extended versions of OFB and CFB are very similar.

Both maintain a one-block shift register X .

The shift register value X_s at stage s depends only on c_1, \dots, c_{s-1} (which are now single bytes) and the master key k .

At stage i , Alice

- ▶ computes X_s according to Extended OFB or Extended CFB rules;
- ▶ computes *byte key* $k_s = L(E_k(X_s))$;
- ▶ encrypts message byte m_s as $c_s = m_s \oplus k_s$.

Bob decrypts similarly.

Shift register rules

The two modes differ in how they update the shift register.

Extended OFB mode

$$X_s = R(X_{s-1}) \cdot k_{s-1}$$

Extended CFB mode

$$X_s = R(X_{s-1}) \cdot c_{s-1}$$

('·' denotes concatenation.)

Summary:

- ▶ Extended OFB keeps the most recent b key bytes in X .
- ▶ Extended CFB keeps the most recent b ciphertext bytes in X ,

Comparison of extended OFB and CFB modes

The differences seem minor, but they have profound implications on the resulting cryptosystem.

- ▶ In eOFB mode, X_s depends only on s and the master key k (and the initialization vector IV), so loss of a ciphertext byte causes loss of only the corresponding plaintext byte.
- ▶ In eCFB mode, loss of ciphertext byte c_s causes m_s and all succeeding message bytes to become undecipherable until c_s is shifted off the end of X . Thus, b message bytes are lost.

Downside of extended OFB

The downside of eOFB is that security is lost if the same master key is used twice for different messages. CFB does not suffer from this problem since different messages lead to different ciphertexts and hence different keystreams.

Nevertheless, eCFB has the undesirable property that the keystreams *are the same* up to and including the first byte in which the two message streams differ.

This enables Eve to determine the length of the common prefix of the two message streams and also to determine the XOR of the first bytes at which they differ.

Possible solution

Possible solution to both problems: Use a different initialization vector for each message. Prefix the ciphertext with the (unencrypted) IV so Bob can still decrypt.

Active adversary attacks

Active adversary

Recall from lecture 3 the active adversary “Mallory” who has the power to modify messages and generate his own messages as well as eavesdrop.

Alice sends $c = E_k(m)$, but Bob may receive a corrupted or forged $c' \neq c$.

How 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.

Some active attacks

Three successively weaker (and therefore easier) active attacks in which Mallory might produce fraudulent messages:

1. Produce valid $c' = E_k(m')$ for a message m' of his choosing.
2. Produce valid $c' = E_k(m')$ for a message m' that he cannot choose and perhaps does not even know.
3. Alter a valid $c = E_k(m)$ to produce a new valid c' that corresponds to an altered message m' of the true message m .

Attack (1) requires computing $c = E_k(m)$ without knowing k .

This is similar to Eve's ciphertext-only passive attack where she tries to compute $m = D_k(c)$ without knowing k .

It's conceivable that one attack is possible but not the other.

Replay attacks

One form of attack (2) clearly *is* possible.

In a *replay* attack, 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 their protocol.

Fake encrypted messages

Even if replay attacks are ruled out, a cryptosystem that is secure against attack (1) might still permit attack (2).

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?

Message-altering attacks

Attack (3) might be possible even when (1) and (2) are not.

For example, if c_1 and c_2 are encryptions of valid messages, perhaps so is $c_1 \oplus c_2$.

This depends entirely on particular properties of E_k unrelated to the difficulty of decrypting a given ciphertext.

We will see some cryptosystems later that do have the property of being vulnerable to attack (3). In some contexts, this ability to do meaning computations on ciphertexts can actually be useful, as we shall see.

Encrypting random-looking strings

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 Alice's 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.

Steganography

Steganography

Steganography, hiding one message inside another, is an old technique that is still in use.

For example, a message can be hidden inside a graphics image file by using the low-order bit of each pixel to encode the message. The visual effect of these tiny changes is generally too small to be noticed by the user.

The message can be hidden further by compressing it or by encrypting it with a conventional cryptosystem.

Unlike conventional cryptosystems, steganography relies on the secrecy of the method of hiding for its security.

If Eve does not even recognize the message as ciphertext, then she is not likely to attempt to decrypt it.

Public-key cryptography

Public-key cryptography

Classical cryptography uses a single key for both encryption and decryption. This is also called a *symmetric* or *1-key* cryptography.

There is no logical reason why the encryption and decryption keys should be the same.

Allowing them to differ gives rise to *asymmetric* cryptography, also known as *public-key* or *2-key* cryptography.

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!
- ▶ e is said to be the *public key* and d the *private key*.

Reason to make e public.

- ▶ Anybody can send an encrypted 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 Bob cannot)!
- ▶ 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 for brute force attacks.

Facts about asymmetric cryptosystems

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 and symmetric cryptosystems are often used together. 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 .
- ▶ Alice computes $c_1 = E_k^1(m)$ and $c_2 = E_e^2(k)$, where e is Bob's public key, and sends (c_1, c_2) to Bob.
- ▶ Bob computes $k = D_d^2(c_2)$ using his private decryption key d and then computes $m = D_k^1(c_1)$.

This is much more efficient than simply sending $E_e^2(m)$ in the usual case that m is much longer than k .

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

RSA

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 why RSA works requires knowing a bit of number theory. 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 $\mathbf{Z}_n = \{0, \dots, n - 1\}$ for some very large integer n .

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

Such a number is unimaginably big. It is bigger than $2^{1023} \approx 10^{308}$.

For comparison, the number of atoms in the observable universe¹ is estimated to be “only” 10^{80} .

¹Wikipedia, https://en.wikipedia.org/wiki/Observable_universe

Encoding bit strings by integers

To use RSA as a block cipher on bit strings, Alice must convert each block to an integer $m \in \mathbf{Z}_n$, 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 $m = c^d \bmod n$.

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

This works because e and d are chosen so that, for all m ,

$$m = (m^e \bmod n)^d \bmod n. \quad (1)$$

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.

²For now, assume all messages and ciphertexts are integers in \mathbf{Z}_n .

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

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

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.

Some number theory

Factoring assumption

The *factoring problem* is to find a prime divisor of a composite number n .

The *factoring assumption* is that there is no probabilistic polynomial-time algorithm for solving the factoring problem, even for the special case of an integer n that is the product of just two distinct primes

The security of RSA is based on the factoring assumption. No feasible factoring algorithm is known, but there is no proof that such an algorithm does not exist.

How big is big enough?

The security of RSA depends on n, p, q being sufficiently large.

What is sufficiently large? That's hard to say, but n is typically chosen to be at least 1024 bits long, or for better security, 2048 bits long.

The primes p and q whose product is n are generally chosen to be roughly the same length, so each will be about half as long as n .

Algorithms for arithmetic on big numbers

The arithmetic built into typical computers can handle only 32-bit or 64-bit integers. Hence, all arithmetic on large integers must be performed by software routines.

The straightforward algorithms for addition and multiplication have time complexities $O(N)$ and $O(N^2)$, respectively, where N is the length (in bits) of the integers involved.

Asymptotically faster multiplication algorithms are known, but they involve large constant factor overheads. It's not clear whether they are practical for numbers of the sizes we are talking about.

Big number libraries

A lot of cleverness *is* possible in the careful implementation of even the $O(N^2)$ multiplication algorithms, and a good implementation can be many times faster in practice than a poor one. They are also hard to get right because of many special cases that must be handled correctly!

Most people choose to use big number libraries written by others rather than write their own code.

Two such libraries that you can use in this course:

1. GMP (GNU Multiple Precision Arithmetic Library);
2. The big number routines in the openssl crypto library.

GMP

GMP provides a large number of highly-optimized function calls for use with C and C++.

It is preinstalled on all of the Zoo nodes and supported by the open source community. Type `info gmp` at a shell for documentation.

Openssl crypto package

OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them.

It is widely used and pretty well debugged. The protocols require cryptography, and OpenSSL implements its own big number routines which are contained in its crypto library.

Type `man crypto` for general information about the library, and `man bn` for specifics of the big number routines.