11. Public-Key Encryption

11. Public-Key (Asymmetric) Encryption

11.1. Overview

Previously we saw symmetric-key encryption, where Alice and Bob share a secret key K and use the same key to encrypt and decrypt messages. However, symmetric-key cryptography can be inconvenient to use, because it requires Alice and Bob to coordinate somehow and establish the shared secret key. *Asymmetric cryptography*, also known as *public-key cryptography*, is designed to address this problem.

In a public-key cryptosystem, the recipient Bob has a publicly available key, his *public key*, that everyone can access. When Alice wishes to send him a message, she uses his public key to encrypt her message. Bob also has a secret key, his *private key*, that lets him decrypt these messages. Bob publishes his public key but does not tell anyone his private key (not even Alice).

Public-key cryptography provides a nice way to help with the key management problem. Alice can pick a secret key K for some symmetric-key cryptosystem, then encrypt K under Bob's public key and send Bob the resulting ciphertext. Bob can decrypt using his private key and recover K. Then Alice and Bob can communicate using a symmetric-key cryptosystem, with K as their shared key, from there on.

11.2. Trapdoor One-way Functions

Public-key cryptography relies on a close variant of the one-way function. Recall from the previous section that a one-way function is a function f such that given x, it is easy to compute $f(x)$, but given y, it is hard to find a value x such that $f(x) = y$.

A *trapdoor one-way function* is a function f that is one-way, but also has a special backdoor that enables someone who knows the backdoor to invert the function. As before, given x, it is easy to compute $f(x)$, but given only y, it is hard to find a value x such that $f(x) = y$. However, given both y and the special backdoor K, it is now easy to compute x such that $f(x) = y$.

A trapdoor one-way function can be used to construct a public encryption scheme as follows. Bob has a public key PK and a secret key SK. He distributes PK to everyone, but does not share SK with anyone. We will use the trapdoor one-way function $f(x)$ as the encryption function.

Given the public key PK and a plaintext message x, it is computationally easy to compute the encryption of the message: $y = f(x)$.

Given a ciphertext y and only the public key PK, it is hard to find the plaintext message x where $f(x) = y$. However, given ciphertext y and the secret key SK, it becomes computationally easy to find the plaintext message x such that $y = f(x)$, i.e., it is easy to compute $f^{-1}(y)$.

We can view the private key as "unlocking" the trapdoor. Given the private key SK, it becomes easy to compute the decryption f^{-1}, and it remains easy to compute the encryption f.

Here are two examples of trapdoor functions that will help us build public encryption schemes:

- *RSA Hardness*: Suppose $n = pq$, i.e. n is the product of two large primes p and q. Given $c = m^e \pmod{n}$ and e, it is computationally hard to find m. However, with the factorization of n (i.e. p or q), it becomes easy to find m.

- *Discrete log problem*: Suppose p is a large prime and g is a generator. Given g, p, $A = g^a \pmod{p}$, and $B = g^b \pmod{p}$, it is computationally hard to find $g^{ab} \pmod{p}$. However, with a or b, it becomes easy to find $g^{ab} \pmod{p}$.

11.3. RSA Encryption

Under construction

For now, you can refer to these notes from CS 70 for a detailed proof of RSA encryption. For this class, you won't need to remember the proof of why RSA works. All you need to remember is that we use the public key to encrypt messages, we use the corresponding private key to decrypt messages, and an attacker cannot break RSA encryption unless they can factor large primes, which is believed to be hard.

There is a tricky flaw in the RSA scheme described in the CS 70 notes. The scheme is deterministic, so it is not IND-CPA secure. Sending the same message multiple times causes information leakage, because an adversary can see when the same message is sent. This basic variant of RSA might work for encrypting "random" messages, but it is not IND-CPA secure. As a result, we have to add some randomness to make the RSA scheme resistant to information leakage.

RSA introduces randomness into the scheme through a *padding mode*. Despite the name, RSA padding modes are more similar to the IVs in block cipher modes than the padding in block cipher modes. Unlike block cipher padding, public-key padding is not a deterministic algorithm for extending a message. Instead, public-key padding is a tool for mixing in some randomness so that the ciphertext output "looks random," but can still be decrypted to retrieve the original plaintext.

One common padding scheme is OAEP (Optimal Asymmetric Encryption Padding). This scheme effectively generates a random symmetric key, uses the random key to scramble the message, and encrypts both the scrambled message and the random key. To recover the original message, the attacker has to recover both the scrambled message and the random key in order to reverse the scrambling process.

11.4. El Gamal encryption

The Diffie-Hellman protocol doesn't quite deliver public-key encryption directly. It allows Alice and Bob to agree on a shared secret that they could use as a symmetric key, but it doesn't let Alice and Bob control what the shared secret is. For example, in the Diffie-Hellman protocol we saw, where Alice and Bob each choose random secrets, the shared secret is also a random value. Diffie-Hellman on its own does not let Alice and Bob send encrypted messages to each other. However, there is a slight variation on Diffie-Hellman that would allow Alice and Bob to exchange encrypted messages.

In 1985, a cryptographer by the name of Taher Elgamal invented a public-key encryption algorithm based on Diffie-Hellman. We will present a simplified form of El Gamal encryption scheme. El Gamal encryption works as follows.

The public system parameters are a large prime p and a value g satisfying $1 < g < p - 1$. Bob chooses a random value b (satisfying $0 \leq b \leq p - 2$) and computes $B = g^b \bmod p$. Bob's public key is B, and his private key is b. Bob publishes B to the world, and keeps b secret.

Now, suppose Alice has a message m (in the range $1 \ldots p - 1$) she wants to send to Bob, and suppose Alice knows that Bob's public key is B. To encrypt the message m to Bob, Alice picks a random value r (in the range $0 \ldots p - 2$), and forms the ciphertext

$(g^r \bmod p, m \times B^r \bmod p)$.

Note that the ciphertext is a pair of numbers, each number in the range $0 \ldots p - 1$.

How does Bob decrypt? Well, let's say that Bob receives a ciphertext of the form (R, S). To decrypt it, Bob computes

$R^{-b} \times S \bmod p$,

and the result is the message m Alice sent him.

Why does this decryption procedure work? If $R = g^r \bmod p$ and $S = m \times B^r \bmod p$ (as should be the case if Alice encrypted the message m properly), then

$R^{-b} \times S = (g^r)^{-b} \times (m \times B^r) = g^{-rb} \times m \times g^{br} = m \pmod p$.

If you squint your eyes just right, you might notice that El Gamal encryption is basically Diffie-Hellman, tweaked slightly. It's a Diffie-Hellman key exchange, where Bob uses his long-term public key B and where Alice uses a fresh new public key $R = g^r \bmod p$ chosen anew just for this exchange. They derive a shared key $K = g^{rb} = B^r = R^b \pmod p$. Then, Alice encrypts her message m by multiplying it by the shared key K modulo p.

That last step is in effect a funny kind of one-time pad, where we use multiplication modulo p instead of xor: here K is the key material for the one-time pad, and m is the message, and the ciphertext is $S = m \times K = m \times B^r \pmod p$. Since Alice chooses a new value r independently for each message she encrypts, we can see that the key material is indeed used only once. And a one-time pad using modular multiplication is just as secure as xor, for essentially the same reason that a one-time pad with xor is secure: given any ciphertext S and a hypothesized message m, there is exactly one key K that is consistent with this hypothesis (i.e., exactly one value of K satisfying $S = m \times K \bmod p$).

Another way you can view El Gamal is using the discrete log trapdoor one-way function defined above: Alice encrypts the message with $B^r = g^{br} \pmod p$. Given only g, p, $R = g^r \pmod p$, and $B = g^b \pmod p$, it is hard for an attacker to learn $g^{-br} \pmod p$ and decrypt the message. However, with Bob's secret key b, Bob can easily calculate $g^{-br} \pmod p$ and decrypt the message.

Note that for technical reasons that we won't go into, this simplified El Gamal scheme is actually *not* semantically secure. With some tweaks, the scheme can be made semantically secure. Interested readers can read more at this link.

Here is a summary of El Gamal encryption:

- **System parameters:** a 2048-bit prime p, and a value g in the range $2 \ldots p-2$. Both are arbitrary, fixed, and public.
- **Key generation:** Bob picks b in the range $0 \ldots p-2$ randomly, and computes $B = g^b \bmod p$. His public key is B and his private key is b.
- **Encryption:** $E_B(m) = (g^r \bmod p, m \times B^r \bmod p)$ where r is chosen randomly from $0 \ldots p-2$.
- **Decryption:** $D_b(R, S) = R^{-b} \times S \bmod p$.

11.5. Public Key Distribution

This all sounds great—almost too good to be true. We have a way for a pair of strangers who have never met each other in person to communicate securely with each other. Unfortunately, it is indeed too good to be true. There is a slight catch. The catch is that if Alice and Bob want to communicate securely using these public-key methods, they need some way to securely learn each others' public key. The algorithms presented here don't help Alice figure out what is Bob's public key; she's on her own for that.

You might think all Bob needs to do is broadcast his public key, for Alice's benefit. However, that's not secure against *active attacks*. Attila the attacker could broadcast his own public key, pretending to be Bob: he could send a spoofed broadcast message that appears to be from Bob, but that contains a public key that Attila generated. If Alice trustingly uses that public key to encrypt messages to Bob, then Attila will be able to intercept Alice's encrypted messages and decrypt them using the private key Attila chose.

What this illustrates is that Alice needs a way to obtain Bob's public key through some channel that she is confident cannot be tampered with. That channel does not need to protect the *confidentiality* of Bob's public key, but it does need to ensure the *integrity* of Bob's public key. It's a bit tricky to achieve this.

One possibility is for Alice and Bob to meet in person, in advance, and exchange public keys. Some computer security conferences have "key-signing parties" where like-minded security folks do just that. In a similar vein, some cryptographers print their public key on their business cards. However, this still requires Alice and Bob to meet in person in advance. Can we do any better? We'll soon see some methods that help somewhat with that problem.

11.6. Session Keys

There is a problem with public key: it is *slow*. It is very, very slow. When encrypting a single message with a 2048b RSA key, the RSA algorithm requires exponentiation of a 2048b number to a 2048b power, modulo a 2048b number. Additionally, some public key schemes only really work to encrypt "random" messages. For example, RSA without OAEP leaks when the same message is sent twice, so it is only secure if every message sent consists of random bits. In the simplified El Gamal scheme shown in these notes, it is easy for an attacker to substitute the message $M' = 2M$. If the messages have meaning, this can be a problem.

Because public key schemes are expensive and difficult to make IND-CPA secure, we tend to only use public key cryptography to distribute one or more *session keys*. Session keys are the keys used to actually encrypt and authenticate the message. To send a message, Alice first generates a random set of session keys. Often, we generate several different session keys for different purposes. For example, we may generate one key for encryption algorithms and another key for MAC algorithms. We may also generate one key to encrypt messages from Alice to Bob, and another key to encrypt messages from Bob to Alice. (If we need different keys for each message direction and different keys for encryption and MAC, we would need a total of four symmetric keys.) Alice then encrypts the message using a symmetric algorithm with the session keys (such as AES-128-CBC-HMAC-SHA-256 [1]) and encrypts the random session keys with Bob's public key. When he receives the ciphertext, Bob first decrypts the session keys and then uses the session keys to decrypt the original message.

[1] That is, using AES with 128b keys in CBC mode and then using HMAC with SHA-256 for integrity