CS 558 Lecture Notes
January 26, 2017

Review and Definitions

- *schemes:* how we do encryption/decryption
- we have a definition of what it means to be secure
- we also have a proof that the scheme satisfies the definition

One Time Pad (OTP) Scheme (for 1 bit messages)

Alice has a secret key *k*, and Bob has the same *k*. To encrypt a message *m* of length 1, you just take the xor of *m* and *k* (which is also of length 1). Alice sends this cipher text *c* to Bob, who then can decipher *c* by taking the xor of *c* and *k* to get the original message *m*.

You can only use the key once in the OTP scheme.

If you want to use the OTP scheme to encrypt a message *m* of length *l,* you need to choose a secret key *k* of length *l*. For each bit in *k,* you then take the xor of it with each bit in *m*, like so:

cipher text *c* = (m1 xor k1, m2 xor k2, … , mL xor kL)

If you use the OTP key twice, an attacker can do a *brute force attack (*where they try every possible key) in just 2 guesses, since in the original OTP scheme, the secret key *k* is of length 1 and there are just two possibilities: 0 and 1. However, if you use an *l*-bit key (secret key of length *l*), the attacker will need to make 2^*l* guesses.

Another thing is that if you simply decide to make the cipher text *c* just the xor of all the smaller cipher bits (where c1 = k xor m1, c2 = k xor m2, and so forth), you will eliminate the need for the key in the first place, as shown below:

c1 xor c2 = (m1 xor k) xor (m2 xor k)
= m1 xor k xor m2 xor k anything xor with itself is 0
= m1 xor m2 xor 0
= m1 xor x2

The above is bad because the key is no longer being used. The randomness in OTP needs to come from the secret key, as messages are not random. If the key is not used and there is the case where parts of the plain text are known, the attacker can then use this to recover the rest.

OTP is not used today, as its not feasible to implement.

Kerckhoff's Principle

2 main things:

- "Enemy knows your system"
- "A crypto system should be secure even if everything about the system is public except for the secret key"

- this means that you should assume that the attacker knows everything about your system (how to encrypt/decrypt) except for the key

attack surface: the parts of a system that are vulnerable to an attack
- we want everything about the scheme to be public so that the attack surface is as small as possible

Stream Ciphers

We start with a random seed *s,* that is typically 128 bits long and is uniformly random, making it so that there are 2^128 possibilities. This seed is fed into a pseudo random generator (PRG) which will essentially spit out an endless stream of pseudo random bits, called a key stream. Note that the PRG is not completely random, which is an important property for it to have. In this scheme, the randomness comes from the seed *s*, and *not* the PRG.

To encrypt a message, it's similar to the OTP scheme in which you have a message *m* of *l* length:

c1 = m1 xor k1
c2 = m2 xor k2

…

cL = mL xor kL

Where k1, k2, …, kL come from the key stream that is generated from the PRG.

If Alice wants to send a message to Bob, she will give the seed *s* to the PRG as an input and will receive a key stream of length *l.* She will then take the xor of each corresponding bit with the bits in her message *m,* and then send the cipher text *c* to Bob. Since the PRG is a deterministic algorithm, using the same input (the seed in this case that both Alice and Bob share) will generate the same output. If the PRG was completely random, Bob would not have the same key stream as Alice and therefore would not be able to decipher her message.

When Bob receives the cipher text *c,* he has to do more or less the same steps as Alice. He will feed the seed *s* into the PRG to get a key stream, and then use the key stream to xor each bit with the corresponding one in the cipher text to get the original message *m.*

The PRG's key stream will appear random if an attacker does not know what the seed *s* is and if *s* is random in the first place. However, if an attacker does know the seed, then they will be able to tell that the generated key stream is not random.

In order for the Stream Cipher to be secure, you must throwaway each key stream after using it. You do not want to reuse bits of the key stream.

Definition of a Secure PRG

Say that there is a polynomial time adversary D, who is currently playing a game in which there are 2 worlds. The first world, world 0, will contain a black box that takes in some random seed *s*,

feeds it into the PRG and then spits out a pseudo random bit string. The second world, world 1, will contain a black box with a single random generator that just spits out a completely random bit string. If the adversary D cannot distinguish which world they are in, then the PRG fits the definition of secure. This is because we want the PRG's bit stream to look like a truly random bit string.

However, D will win the game each time if they have the seed *s,* since they could just plug in *s* to a PRG algorithm and then compare that output with each output in the world. If they're the same, then D can distinguish which world they are in and therefore the PRG is *not* secure.

Some examples of PRGs that are used are:
- RC4
- Salsa
- Chacha

Block Cipher

A block cipher is a pseudo random permutation, where a permutation is a one-to-one mapping such that each input is mapped to one and only one output and vice versa. How a block cipher works is that a key that represents a permutation is fed into the PRP, and then an element into the input set will be fed to it as well which will lead to the PRP outputting the element in the output set that it should correspond to.

It is important to also note that if you have a set of size n, the number of possible permutations is n!. So if you have an input set of 128 bit strings, there are 2^128 possible inputs/outputs (since permutation is a one-to-one mapping). This means that with there are also (2^128)! possible permutations! Fortunately for the block cipher, you only choose one of them.