CMSC 28400 Introduction to Cryptography **Autumn 2021**

Notes #6: Block Ciphers

Instructor: David Cash

Along with stream ciphers, *block ciphers* are a very widely-deployed type of algorithm used for encryption and other cryptographic purposes. These notes cover the basic notion of a block ciphers and some interesting attacks. In the next notes we'll learn about how block cipher are put together to build larger algorithms, such as encryption of large files or tools for authentication.

6.1 Introduction to Block Ciphers

Let's start with the definition.

Definition 6.1. *A cipher $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ is called a* block cipher *if $\mathcal{M} = \mathcal{C} = \{0,1\}^\ell$ for some positive integer ℓ. The integer ℓ is called the* block length *or* block size *of E.*

When E is a block cipher, then for each $k \in \mathcal{K}$, $E(k, \cdot)$ must actually be computing a *permutation* on $\{0,1\}^\ell$. In practice we will always have $\mathcal{K} = \{0,1\}^n$ for some positive integer n, and we say that n is the *key-length* of E.

We will sometimes want to evaluate E^{-1}, the decryption algorithm. Thus all practical block ciphers, like DES and AES that follow, support efficient decryption.

Example 6.1. *The DES blockcipher, described in Boneh-Shoup Chapter 3 and many other places, is a function*

$$\text{DES} : \{0,1\}^{56} \times \{0,1\}^{64} \to \{0,1\}^{64}.$$

The key length is 56 *and the block length is* 64*. As we will see, both of these lengths are a major limitation.*

The replacement for DES is a family of three blockciphers, often called AES128, AES192, and AES256, which have a common block length of 128 *and key lengths of* 128, 192*, and* 256 *respectively. The* 128 *bit version is the most commonly used. Later we'll refer back to this as simply* the *AES blockcipher, with the notation*

$$\text{AES} : \{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}.$$

We do not recall the structure of the AES block ciphers here, but you can read about them in Boneh-Shoup as well.

The construction of DES, AES, and other commonly-used block ciphers like KASUMI is an deep and active area of cryptography. It is also highly specialized, and unfortunately in a one-quarter introduction we can't do everything. In addition to Boneh-Shoup, you can also check out *The Block Cipher Companion* by Knudsen and Robshaw, which is available as a pdf from the library: `https://catalog.lib.uchicago.edu/vufind/Record/8899690`.

6.1.1 Security of Block Ciphers

What does it mean for a block cipher $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ to be *secure*? Or, *not secure*? For a stream cipher $G : \{0,1\}^n \to \{0,1\}^\ell$ we arrived at the notion of pseudorandom generators and distinguishing advantage, which intuitive qualified the G to be used in place of a one-time pad. For a block cipher however this definition does not fit; It's not even clear what we should give an adversary.

In the next set of notes we will look at a formal a notion called *pseudorandom permutation distinguishing advantage*. That definition will probably make more sense after we look at how block ciphers are used. Just like the pseudo-one-time-pad motivated the pseudorandom generator notion, applications of block ciphers will drive their security requirements.

For now we'll point out some informal requirements. If we plan to encrypted blocks with E, then it in almost any imaginable scenario we would want it to be hard to recovery the blocks we are encrypting. In a bit more detail, suppose we are using a key $k \in \{0,1\}^n$ and sending $c_i = E(k, m_i)$ for several blocks $m_1, m_2, \ldots \in \{0,1\}^\ell$. If an adversary captures $c_1, c_2, \ldots$, we might ask that it be hard to recover $m_1, m_2, \ldots$. This is called a *known-ciphertext message recovery attack*, so called because the adversary knows the ciphertexts and is targeting the input message blocks.

An even more basic requirement is that the *key* k be difficult to recover from the blocks $c_1, c_2, \ldots$; call this *known-ciphertext key recovery attack*. If an adversary can recover k then it can certainly recover $m_1, m_2, \ldots$, but it is possible in principle that it finds a shortcut to recover the message blocks without recovering the k; This means security against known-ciphertext *key*-recovery attacks is *weaker* than security against known-ciphertext *message* recovery attacks.

We could go on like this for a while, thinking about different combinations of what an adversary is given and what it is trying to find. We'll stop with one more example that will drive a case-study enlightening some interesting issues in the next section.

It turns out that very often an adversary learn not only some $c_1, c_2, \ldots$, but also the input message blocks $m_1, m_2, \ldots$. Perhaps it doesn't learn every single input message block (otherwise nothing is really hidden), but in practice adversaries can often obtain some blocks like this. For instance, if a web server is encrypting web pages, then adversary will very often know that particular input blocks correspond to protocol boiler-plate text; In that case it can match up some blocks c_i with blocks m_i that it knows.

This called a *known plaintext attack*. Abstractly, we'll assume an adversary is given some "examples" $(m_1, c_1), (m_2, c_2), \ldots$ where $c_1 = E(k, m_1), c_2 = E(k, m_2), \ldots$, all computed under the same unknown key k. As a natural goal for the adversary is to recover k, so that it can decrypt future blocks as they're sent. This is a *known-plaintext key-recovery attack*, which is the subject of the next two sections.

6.2 Exhaustive Key Search

Consider the setting for a known-plaintext attack that has captured some message/ciphertext pairs $(m_1, c_1), (m_2, c_2), \ldots$ where $c_i = E(k, m_i)$. A simple strategy to find k is simply try every possible key and see which one "works". More formally, it could work as follows:

INPUT: Examples $(m_1, c_1), (m_2, c_2), \ldots$, where $c_i = E(k, m_i)$
OUTPUT: A key $\hat{k} \in \mathcal{K}$
For $\hat{k} \in \mathcal{K}$:
    If $E(\hat{k}, m_i) = c_i$ for all i: Output $\hat{k}$

There are two primary properties to consider: Correctness and runtime. The algorithm will always terminate and output *some* $\hat{k} \in \mathcal{K}$, because it will at least stop when it gets the correct key. However it may in principle output another key that also maps all of the m_i to the respective c_i. In practice this basically never happens once more than a few examples are happened; We will ignore this possibility and assume the algorithm is correct for our purposes.

For run time, we are mostly concerned with the number of iterations of the "For" loop, which is $|\mathcal{K}| = 2^n$ for a block cipher with n-bit keys. The DES block cipher only has 2^{56} keys, which is not very large by modern standards, and today even a modestly-equipped adversary can execute this algorithm quickly (hours or minutes depending on their computer). Against AES, the runtime is 2^{128}, which is generally considered totally infeasible.

6.3 Double Encryption

DES was retired because of its small key length and block length, but many systems and protocols were deployed with DES baked into them (ATMs are a famous example). Instead of completely rebuilding the systems that used DES to convert them to a block cipher with larger parameters like AES, many opted to use *triple encryption* where they ran DES three times instead of two.

In this section we investigate their interesting decision to encrypt three times rather than two. Here are the options considered, which we denote 2DES and 3DES. "Double DES" has the form

$$\text{2DES} : \{0,1\}^{112} \times \{0,1\}^{64} \to \{0,1\}^{64}.$$

On input $k \in \{0,1\}^{112}$ and $m \in \{0,1\}^{64}$, 2DES will divide the key into two 56 bit keys k_1 and k_2 and outputs

$$\text{2DES}(k_1 \| k_2, m) = \text{DES}(k_2, \text{DES}(k_1, m)).$$

You can check that 2DES satisfies the definition of a block cipher. The key length of 2DES is 112, and an exhaustive key search would run on the order of 2^{112} time, which is likely infeasible. It would appear that 2DES is strong enough for practice. However, due to the interesting "meet in the middle" attack that we look at next, 2DES is not used; Instead three-key versions are used, like

$$\text{3DES} : \{0,1\}^{168} \times \{0,1\}^{64} \to \{0,1\}^{64}$$

which maps

$$\text{3DES}(k_1 \| k_2 \| k_3, m) = \text{DES}(k_3, \text{DES}(k_2, \text{DES}(k_1, m)))$$

where k_1, k_2, k_3 are 56 bits each. In order to save on key storage, sometimes a variant called 3DES2 is used, where

$$\text{3DES2} : \{0,1\}^{112} \times \{0,1\}^{64} \to \{0,1\}^{64}$$

and

$$\text{3DES2}(k_1 \| k_2, m) = \text{DES}(k_1, \text{DES}^{-1}(k_2, \text{DES}(k_1, m))).$$

This version seems to avoid known attacks. Note the middle application of DES is actually a decryption operation; This allows for an easy fallback mode, where if one sets $k_1 = k_2$ then 3DES2 will collapse back to "single" DES.

6.3.1 Meet-in-the-Middle Attack on Double Encryption

We now give the attack that breaks 2DES with about 2^{56} computation. It is a known-plaintext key recovery attack, that works well with only three or more known-plaintext blocks. To some approximation, it shows that that 2DES is no better than single DES! We are looking into this because it is an elegant method, and also because it highlights the subtleties of security.

The attack is called *meet-in-the-middle*, and we give it shortly. This algorithm crucially uses a *hash table* with some assumed properties; We won't get into exactly how the table is implemented, because it's pretty standard and would involve quite a bit of detail that is not our focus. We will just assume the table H gives us the following capabilities:

- Given any bit strings x and v, we can associate v with x in constant time. We write this as $H[x] \leftarrow v$.
- Given a bit string x, we can look up the string associated with x, or detect that none exists, in constant time. We write $H[x]$ for the string associated with x; if there is none, we express this by saying $H[x] = \perp$.

Intuitively, H is like an array, except we can use strings as indexes rather than numbers in some range. If you've used Python dictionaries, they provide essentially this interface.

Now for the meet-in-the-middle attack:

INPUT: Examples $(m_1, c_1), (m_2, c_2), \ldots, (m_t, c_t)$ where $c_i = \text{2DES}(k_1 \| k_2, m_i)$
OUTPUT: A key $\hat{k}_1 \| \hat{k}_2 \in \{0,1\}^{112}$
Initialize a hash table H
For $\hat{k}_1 \in \{0,1\}^{56}$:
    $x \leftarrow \text{DES}(\hat{k}_1, m_1) \,\|\, \cdots \,\|\, \text{DES}(\hat{k}_1, m_t)$
    $H[x] \leftarrow \hat{k}_1$
For $\hat{k}_2 \in \{0,1\}^{56}$:
    $x' \leftarrow \text{DES}^{-1}(\hat{k}_2, c_1) \,\|\, \cdots\cdots \,\|\, \text{DES}^{-1}(\hat{k}_2, c_t)$
    If $H[x'] \neq \perp$:
        $k_1' \leftarrow H[x']$; $k_2' \leftarrow \hat{k}_2$
        Output $k_1' \,\|\, k_2'$

What's going on in this algorithm? It is based on the following observation: For the correct key $k_1 \| k_2$, it holds that

$$(\text{DES}(k_1, m_1), \ldots, \text{DES}(k_1, m_t)) = (\text{DES}^{-1}(k_2, c_1), \ldots, \text{DES}^{-1}(k_2, c_t)).$$

So we can compute every possible value on each side of the equation and then search for the collision (the "meeting in the middle"). The trick is that we are reusing our computational effort here. Instead of trying each combination k_1 and k_2 together, we are able to reuse a single evaluation with a guess for k_1 for *every* guess of k_2.

The attack runs in time about $2 \cdot 2^{56} = 2^{57}$, which is approximately equal to the effort required to break single DES.

Exercise 6.1. *Let's estimate how the value of t affects the probability that we get the correct key. The probability can't go down as t gets larger, but larger t result in more computational effort, so we prefer to use some minimal value.*

Actually computing the correct value of t would depend on the inner workings of DES. *Instead we'll do it heuristically. Assume that all of the entries of H are* uniformly random and independent strings*, except for the entries that correspond to k_1 and k_2, which are equal. Using the union bound (see the probability notes), give a simple upper bound on the probability that the algorithm would output an incorrect key pair under this heuristic.*

Exercise 6.2. *Adapt the meet-in-the-middle attack to* 3DES. *How long does your attack run, and how much memory does it consume? Repeat the Exercise 6.1 to estimate the number of examples required to get a good bound on the probability your algorithm will output the wrong key.*

Exercise 6.3. *Consider the block cipher $E : \{0,1\}^{112} \times \{0,1\}^{64} \to \{0,1\}^{64}$ defined as*

$$E(k_1 \| k_2, m) = \mathrm{DES}(k_1, m) \oplus k_2,$$

where $k_1, k_2 \in \{0,1\}^{56}$. Find a known-plaintext key recovery against E that is not much more expensive than attacking plain DES. *Repeat exercise 6.1, but with your new attack.*

Exercise 6.4. *Consider the block cipher $E : \{0,1\}^{168} \times \{0,1\}^{64} \to \{0,1\}^{64}$ defined as*

$$E(k_1 \| k_2 \| k_3, m) = \mathrm{DES}(k_1, m \oplus k_3) \oplus k_2,$$

where $k_1 \in \{0,1\}^{56}$ and $k_2, k_3 \in \{0,1\}^{64}$. Find a known-plaintext key recovery against E that takes about 2^{56+64} time. Repeat exercise 6.1, but with your new attack.