CS 198 Fall 2022 | Codebreaking | Ryan Cottone, Imran Khaliq-Baporia | Note 9

Block Ciphers

As we have seen with hashing, many cryptographic schemes are much easier to build when we consider fixed-length *blocks* of inputs to do operations on. Block ciphers operate this way, taking in some input, splitting it up into blocks, encrypting each block, and recombining afterward. We will primarily be focusing on AES (Advanced Encryption Standard), which is the premier block cipher in use today. AES is so secure that even the NSA approves its use for top-secret data. No (practical) attacks are known on correctly-implemented AES as of 2022 [1].

The specifics of the cryptography within the actual "operation" is out of scope for this class. You can think of it like a function $f(k,d) = c$ taking in a fixed-length key k and data d of the same length, where c is the output (again of the same length). The actual operation within these blocks is considered the core "algorithm", whereas how we *use* it is the focus of modes of operation.

Modes of Operation

We now have our block cipher function $f(k,d)$, but how do we use it to encrypt arbitrarly long data? First, we split the data up into blocks $b_1, b_2, \ldots b_k$, padding as required. Once we have these, we must figure out how to order our encryption.

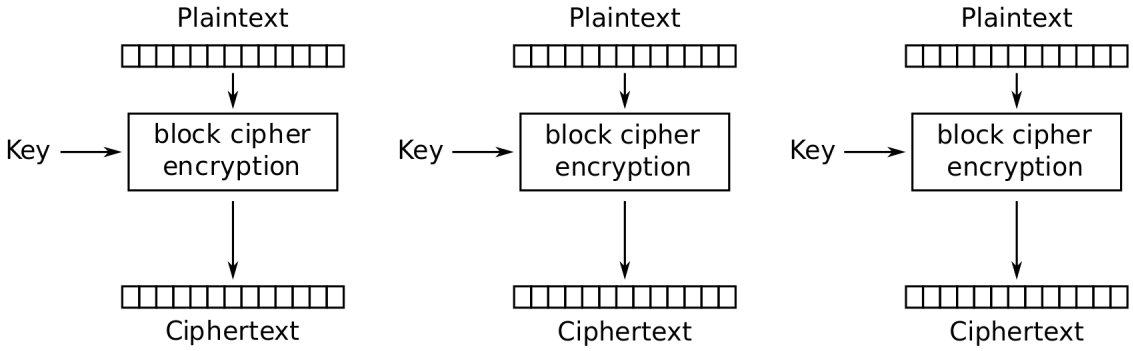
The most straightforward way of encrypting might be to set $c_i = f(k,b_i)$ for all $i = 1,\ldots,k$. In effect, we encrypt every block individually with the same key and output the ciphertext blocks respectively. Assuming that f is secure, this won't immediately *leak* our encrypted message, but it falls prey to a weaker attack. Say we were commanding some invasion force, monitoring some early-warning radar station. Every 15 minutes, the station broadcasts an encrypted message "ALL CLEAR". If they spot us, they will instead broadcast some other message. We only need to know when we've been spotted. Can you think of why we can easily do this?

The reason is that this AES mode is entirely *deterministic*, in that encrypting the same message will always output the same ciphertext. So if we ever see a different message than has been sent before, we know we've been spotted. Even worse, we can tell exactly which blocks in the message have changed. If we only changed b_3, then only c_3 will have changed in the output!

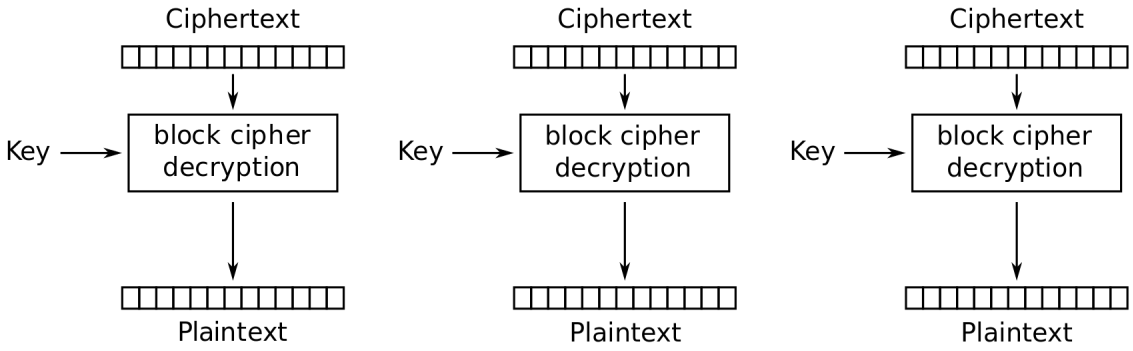
Formally, this is known as the loss of IND-CPA (indistinguishable under chosen-plaintext attack) security, and is a very bad thing to not have in a cryptosystem (you will likely be familiar with this if you have taken CS 161). The mode we have been describing is also known as AES-ECB (Electronic Code Book).

[1] There is a largely theoeretical attack named the *biclique attack* that shaves a few bits off the keysize, but it is both infeasible to implement and not worth the tradeoff.

A visual aid for how ECB works (courtesy of CS 161 textbook):



Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

AES-CBC, or Cipher Block Chaining, is a secure (and widely used) mode of operation for AES. Instead of blindly applying the key linearly, it utilizes the result of the previous encryption to act as a source of "randomness" for the next one (note it isn't actually random). At any given step i, we have to know C_{i-1}, P_i, and k to compute C_i. The formula is:

$$C_i = \text{Enc}(k, C_{i-1} \oplus P_i)$$

We can reverse this to find the decryption formula:

$$P_i = \text{Dec}(k, C_i) \oplus C_{i-1}$$

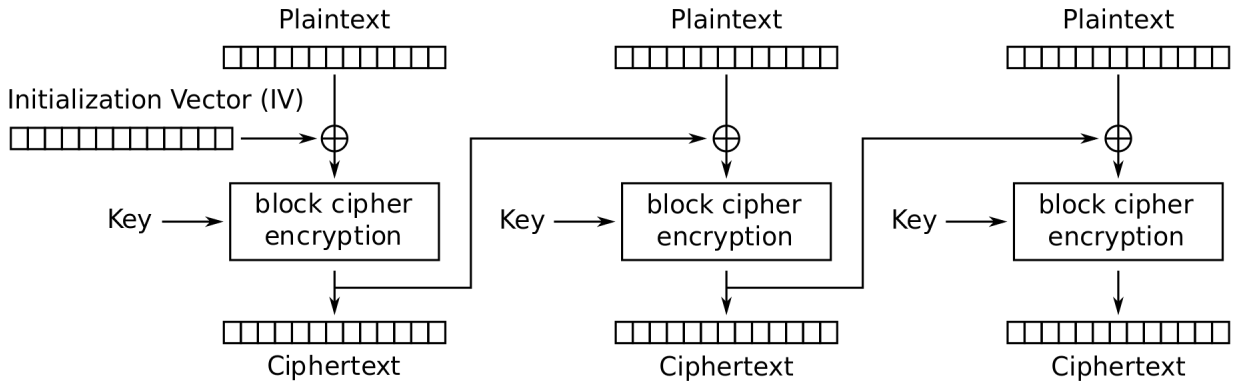
Informally, we encrypt the XOR of the previous ciphertext block and the current plaintext block with the key. This means any difference in the previous block would "carry forward" and change the results of the future blocks. Remember that the output of a good block cipher is (practically) indistinguishable from random noise, so even a small random change in block one would be enough

to make the entire ciphertext look totally different.

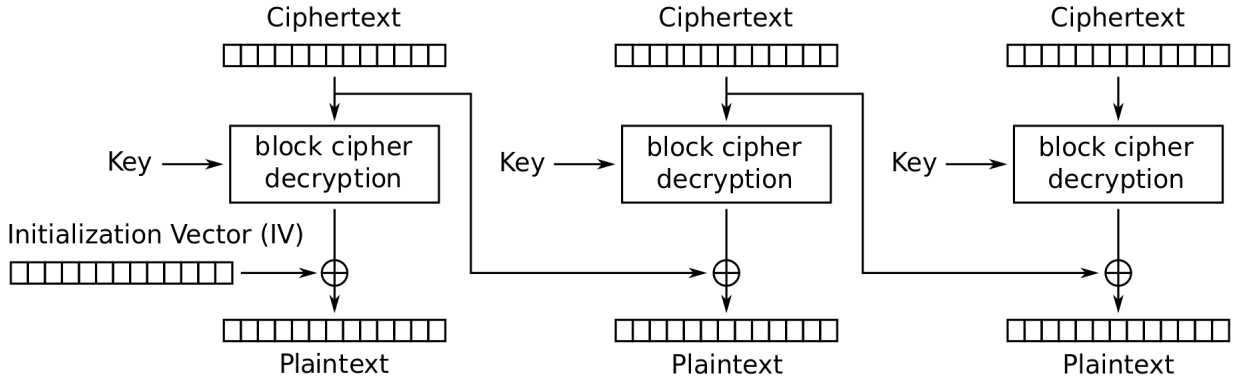
This raises the question – how do we encrypt C_1? We would need the previous ciphertext, but that doesn't exist. Enter the concept of the **initialization vector**, a fancy term for a "ciphertext" block that we provide to start the encryption. **In order for AES-CBC to be secure, this IV must be randomly generated each time we send a new message!** Can you see why AES-CBC might not be IND-CPA secure if we re-use the IV?

Note that the receiver must also know the IV to decrypt the message. For this reason, IVs are considered to be public knowledge, since we send $[IV, C_1, C_2, \ldots C_k]$ as the message. This does not affect security, however (assuming you didn't reuse the IV)!

Here is a visual aid of AES-CBC, again courtesy of the CS 161 textbook:



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

It turns out we can decrypt every block at once since we know the entirety of C, but encrypting is a painfully sequential process.

Stream Ciphers

Another popular type of cipher is the **stream cipher**, which operates on variable lengths of data that come in as a "stream". You can think of it like encrypting a phone call in real-time – we don't know the length of data beforehand. We could try to use a block cipher, but this can get messy since it requires data in fixed-length blocks before we send it over.

If we had a long, secure bitstring, we could just XOR the plaintext with that and send it immediately (exactly like the one-time pad). As we saw with the one-time pad, however, this is a bit infeasible. Instead, what if we use the output of a CSPRNG as the bit string? Sure, this is not ideal, but if the PRNG output is reasonably good, it will be good enough.

This is the basic idea behind many popular stream ciphers like RC4. Though, RC4 is considered broken nowadays due to the output leaking a significant amount of information about the key itself (attacks which are unfortunately out of scope for this class).

Stronger stream ciphers such as ChaCha20 are mainly in use today. They have much more complicated internals than just a basic PRNG, but the core idea is roughly the same.

Contributors:

- Ryan Cottone