Encryption algorithms

For our algorithm, we are given a choice between asymmetric and symmetric ciphers. Asymmetric schemes solve the problem of needing separate keys for every connection. However, compared to symmetric ciphers, asymmetric ciphers are generally quite slow and require larger key sizes. As such, we should instead use asymmetric ciphers to establish a secure channel to exchange symmetric cipher keys. Using RSA is sufficient for this purpose. In our implementation, we will pretend that a certificate authority has already authenticated the public key. Since implementing a certificate authority would require a third party to be involved.

The Symmetric blockciphers available for us to implement are 3DES, Blowfish, and AES (Rijndael). 3DES is an improvement to DES, done by performing DES three times. It uses a block size of 64 bit and its strongest variant has a key length of 168 bits. However, due to some clever brute forcing optimization tricks, its effective bit security is only 112 bits. In other words, instead of checking all possible combinations in a 168 bits key, it is sufficient to just check an equivalent number of combinations in a 112 bits key. Patil et al. (2016) shows that in terms of performance, 3DES is quite slow.

Blowfish has a variable key size of 32-448 bits. Patil et al. (2016) shows that Blowfish performs much faster than 3DES. However, before Blowfish can encrypt and decrypt messages, it requires a preprocessing stage on the used key, and is very slow and computationally expensive (Schneier, 1994).

Both 3DES and Blowfish use a small block size of 64 bits and are vulnerable to birthday attacks. Which makes them insecure given enough messages are observed by an adversary. The Successor to Blowfish, Twofish fixed this by increasing the block size, however it's not implemented in both our java and python cryptography library that we chose.

AES, the successor to DES, uses a block size of 128 bits. This significantly reduces the effectiveness of birthday attacks. Patil et al. (2016) also shows that AES performs much faster than 3DES, Albeit not as fast as Blowfish. Additionally, lots of modern CPUs are adding dedicated instructions for AES, which makes AES perform even better (Shay Gueron, 2010).

Blowfish could be a suitable candidate if we desire its performance. To reduce the associated risk with birthday attacks, we could refresh our keys occasionally. However, that would be counter-productive as we'll have to re-do the slow pre-processing step again.

That leaves us with AES as the choice for our implementation. AES comes with key sizes of 128-bit, 192-bit, and 256-bit. Higher key sizes yield better security, but at the cost of performance. Since there are no known optimization tricks for brute-forcing AES, key sizes of n-bits will still have n bits of security. Assuming an AES key size of 128-bit, an attacker would have to go through 2^128 combinations to guess the key. We regard that as strong enough and will use a 128-bit key in our implementation.

Encryption Mode of Operation

For our encryption modes of operation, we have ECB, CBC, CTR, CFB, and OFB Available in both our Java and Python library. Notably, some of them will require padding to fit within a multiple of AES's block size, while others will transform our blockcipher into a streamcipher-like mode of operation.

ECB stands for Electronic Code Book. It is the simplest mode of operation. Essentially, we're individually encrypting our message block-by-block. If we reach the end of the message and there aren't enough bits to form a block, we add dummy bits (padding) to complete the block. As demonstrated by Figure 1 and Figure 2, This mode has a side effect of identical blocks encrypting into identical ciphertext. Using this mode of operation, we could risk attackers observing patterns among the ciphertext blocks.

CBC (cipher-block-chaining) mode fixes the above issue with ECB. Before encryption, It XORs a plaintext block with the previous ciphertext block. In the case of the first block, it XORs the first block with an IV that will be transmitted alongside the ciphertext (see Figure 3 & 4). However, by using this mode, we lose the ability to encrypt in parallel- as encrypting a block will require the previous block to be already encrypted. Furthermore, it should be noted that this mode suffers from the padding oracle attack when used with MAC-then-Encrypt.

CFB (cipher feedback) mode is similar to CBC, in that the encryption of a block involves the previous ciphertext block. However, it is also wildly different as it transforms AES into a streamcipher-like mode of operation. it encrypts the previous ciphertext block and uses that as a keystream for XORing the plaintext (see figure 5 & 6). This mode also has the same downside as CBC mode. Since encrypting a block will still require the previous block to be already encrypted, the encryption phase cannot be parallelized.

OFB (output feedback) mode is another way of generating keystreams from a blockcipher. In contrast with CFB mode, instead of encrypting the previous ciphertext block, a keystream is generated by encrypting the previous block's keystream (see figure 7 & 8). As a result, encryption/decryption of a block will require the previous block's keystream to be already calculated. Thus, both the encryption phase and decryption phase cannot be parallelized

CTR (Counter) mode is another streamcipher-like mode of operation. It encrypts an incrementing counter to generate the keystream (see figure 9 & 10). The two communicating parties will agree on an initial counter value using a nonce/IV. Since the keystreams are only reliant on the counter value, we could skip to specific counter value and start the encryption/decryption process from an arbitrary location. As such, this mode is highly parallelizable.

Aside from ECB mode, there are no obvious security flaws with all the other modes of operation. Therefore, we will use CTR mode for its performance. However, we must be careful with the nonce/IV. For CTR mode specifically, the same counter value must not be repeated under the same key (Wiley, 2012, p. 71). A simple and sufficient implementation would be to initialize the counter at 0 for every session, each successive message will start their counter with the immediate value of the last counter of the last message. When compared to randomly selecting an IV, initializing the counter at 0 offers better collision resistance and allows for more data to be encrypted before switching to a new key. Furthermore, incrementing the counter progressively throughout the session offers inherent protection against replay-attacks, receiving a ciphertext meant for the past will result in the ciphertext being decrypted with a higher counter value, making it a "garbled mess".

MAC

Notably, our chosen mode of operation only assures confidentiality. Leaving our recipient clueless as to whether the message has been tampered with or not. This is where Message Authentication Codes (MACs) come in. MACs guarantee the authentication of the message and thus the integrity. MACs will also be important for making the data chosen cipher-attack (CCA) secure, as to infer data from the ciphertext, they must be able to bypass the MAC.

To elucidate, MACs are essentially a cryptographic checksum that generate alongside the message. In doing so, they confirm that the contents of the message haven't been altered with, preserving the data's integrity and authenticity. This is because if the user alters the message, without altering the MAC, then the recipient will receive a different MAC, compared to the original MAC, meaning they know it has been altered. Essentially MACs allow us to confirm if the message has been modified. For our MAC method, we had three different techniques to choose from, each with their own advantages/disadvantages.

The first method is Cipher-based Message Authentication Code (CMAC), which is a variation of the MAC method, CBC-MAC. To elucidate, CMAC works by dividing the message into blocks each with an equal amount of information within them, then those blocks are encrypted with the last block being treated differently, by XORing one of two special values in that last block (Ferguson et al., 2010b). In doing so, CMAC allows us to check for not only accidental modifications, but intentional tampering as well, guaranteeing message integrity and authenticity.

Galois Message Authentication Code (GMAC) is another MAC method, which uses a nonce, the message and a key. With the nonce also needing to be known by the recipient. GMAC works by computing a mathematical function on the plaintext, which is then used to create a ciphertext with a block cipher in CTR mode. The nonce is then used as the IV for the CTR, producing the MAC.

The third method is Hash-based message authentication (HMAC) which uses hash functions to form the MAC. Unlike CMAC, which encrypts each block separately, HMAC works by using a private key and a hash function with an input to produce our MAC. To elaborate, if we use HMAC and provide the same message and hash function but use two different private keys, we will produce two different MACs. Meaning that as long the attacker does not have the private key, we will be able to know who the message came from providing authenticity, and thus know if it has been tampered with, meaning we have plaintext integrity.

Now while HMAC and CMAC achieve the same goal of providing both authenticity and integrity, the main benefit of using HMAC is that it is faster than CMAC. To elaborate, CMAC takes each individual block and encrypts them separately, whilst HMAC uses a hash function. Conversely, CMAC can become a faster/efficient approach if the environment allows for it, that being the system is set up so that it has hardware acceleration for block ciphers. Albeit HMAC and CMAC achieve both integrity and authenticity, hashing functions are generally faster than symmetric block ciphers.

As for GMAC, it has a huge flaw due to the requirement of a nonce. By using GMAC, we would have to encrypt our blocks that are all zero with the AES key (in ECB mode) and then store it in the variable H (nonce and key). H is then used to authenticate ALL the plaintexts that have been authenticated by the AES key (Soatok, 2022). The problem here is that the nonce can be reused repeatedly by the attackers to find H which is used to authenticate the plaintexts, which would violate the integrity and authenticity of the data. However, it stops there. If they repeatedly find H, theoretically the attackers could keep abusing it to tamper with messages for as long as they have the nonce. That is but one example, called the "forbidden attack" in which GMAC fails to guarantee authenticity and integrity. Ultimately, GMAC will not be used, as by providing a nonce we will be risking modifications to the ciphertext.

Thus, we will be opting for HMAC as it is more secure than GMAC and a faster approach than CMAC. The downside/benefit to HMAC is that the strength of its security depends on the properties of the hashing algorithm used. For example, the hashing algorithm MD5 has been shown to be weak to collision search attacks (Wang & Yu, 2005) but is faster than the hashing algorithm SHA-1. However, SHA-1 is much more secure than MD5, despite performing slower. Ultimately, HMAC is the MAC of choice as it performs faster than the other two and depending on the hashing algorithm used may be equally or more secure than the other two MAC algorithms.

Appendix

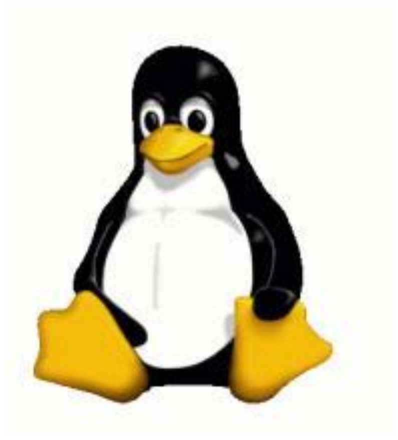

Figure 1: Tux the penguin drawn by Larry Ewing using the GIMP software (https://isc.tamu.edu/~lewing/linux/)

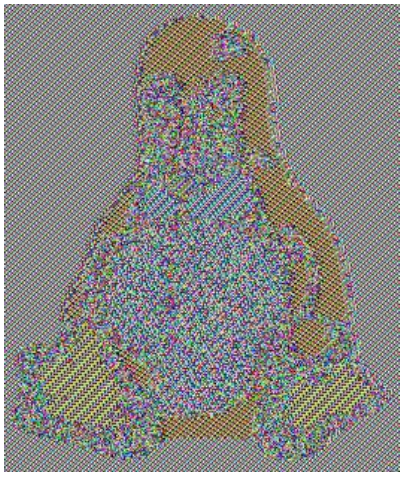

Figure 2: ECB-encrypted Tux generated by GitHub user robertdavidgraham (https://github.com/robertdavidgraham/ecb-penguin)

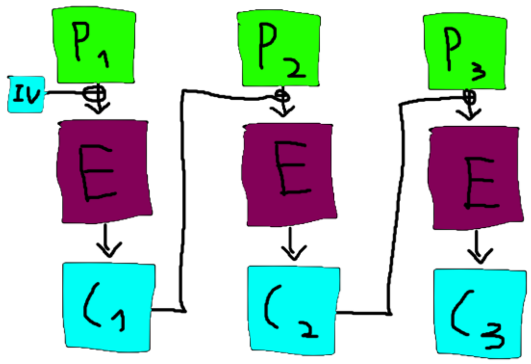

Figure 3: CBC encryption

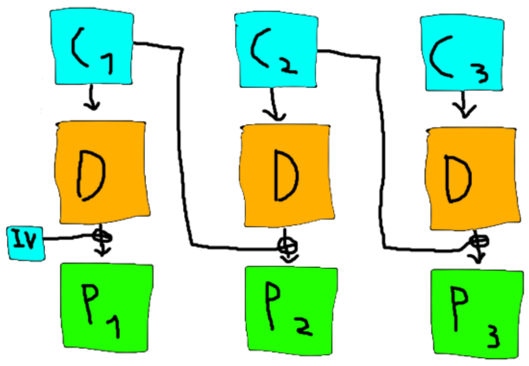

Figure 4: CBC decyption

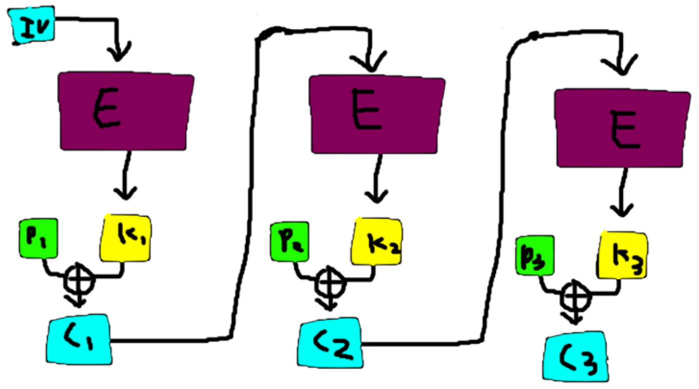

Figure 5: CFB encryption

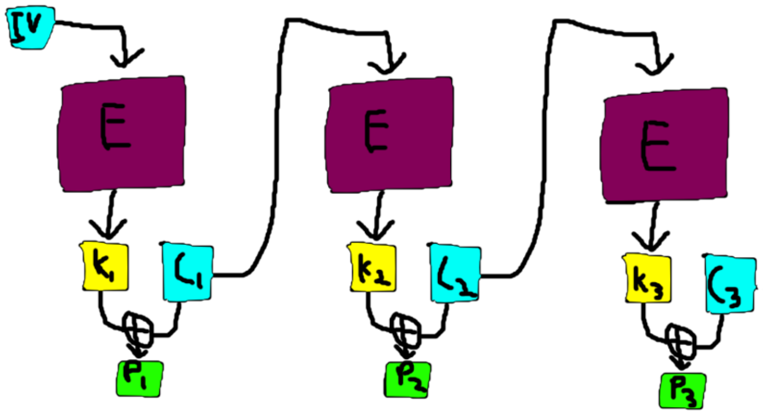

Figure 6: CFB decryption

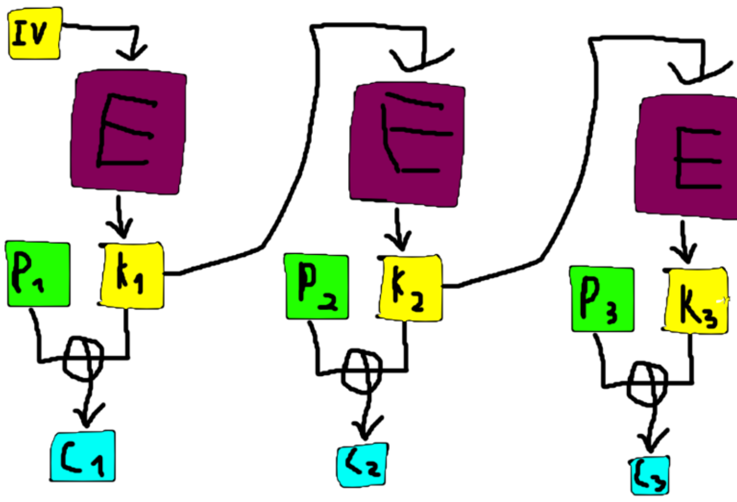

Figure 7: OFB encryption

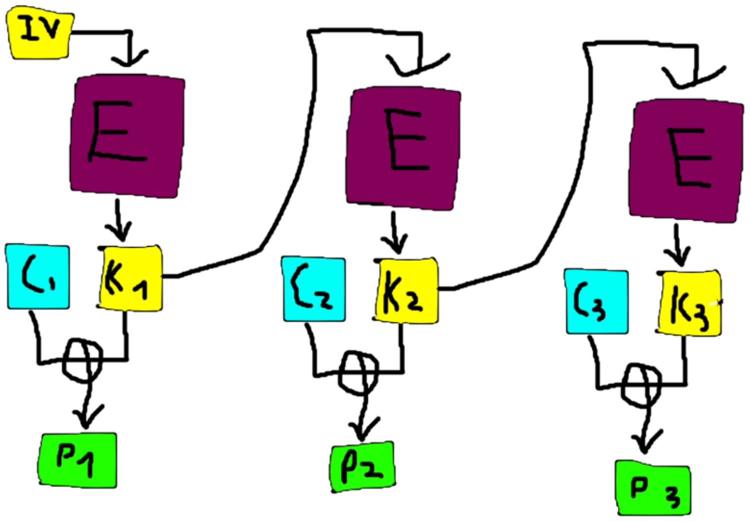

Figure 8: OFB decryption

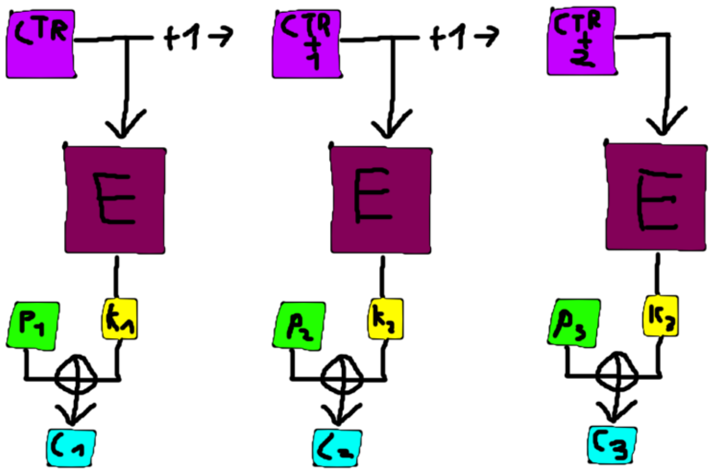

Figure 9: CTR encryption

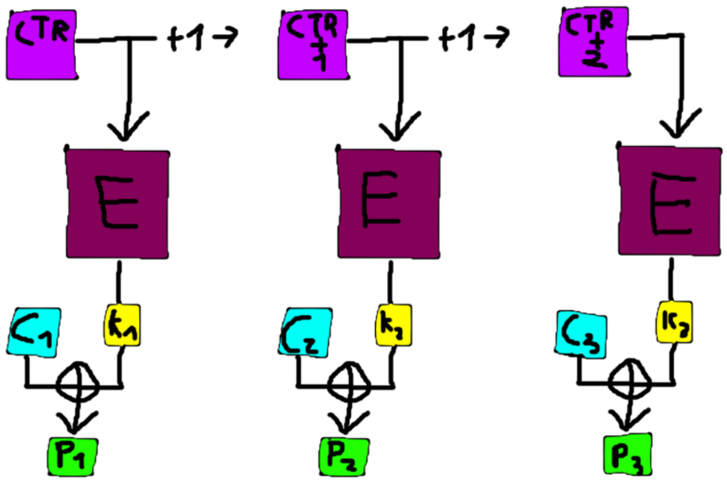

Figure 10: CTR decryption

References

1. Patil, P., Narayankar, P., Narayan D.G., & Meena S.M. (2016). A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish. Procedia Computer Science, 78, 617–624. https://doi.org/10.1016/j.procs.2016.02.108
2. Schneier, B. (1994). Description of a new variable-length key, 64-bit block cipher (Blowfish). Fast Software Encryption, 191–204. https://doi.org/10.1007/3-540-58108-1_24
3. Shay Gueron. (2010, May). Intel ® Advanced Encryption Standard (AES) New Instructions Set (Revison 3.0). In Intel.com (No. 323641–001). Retrieved October 22, 2022, from https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf
    a. Ferguson, N., Schneier, B., & Kohno, T. (2010b). *Cryptography Engineering: Design Principles and Practical Applications* (1st ed.). Wiley.
4. Obe, B. B. (2022, February 12). *I Know HMAC, But What's CMAC? - ASecuritySite: When Bob Met Alice*. Medium. https://medium.com/asecuritysite-when-bob-met-alice/i-know-hmac-but-whats-cmac-b859799af732
5. Soatok, V. A. (2022, August 24). *Why AES-GCM Sucks*. Dhole Moments. https://soatok.blog/2020/05/13/why-aes-gcm-sucks/

6. Wang, X., & Yu, H. (2005). How to Break MD5 and Other Hash Functions. *Lecture Notes in Computer Science*, 19–35. https://doi.org/10.1007/11426639_2
7. Wiley. (2012). *Cryptography Engineering: Design Principles and Practical Applications*.