Applied Cryptography - Final project report on AES implementation

Arizona State University, Spring Term 2021, CSE 539

Kuntal K. Nayak
Arizona State University
knayak3@asu.edu

***Abstract*—Block Cipher is a Symmetric Key Cipher that operates on blocks of a fixed number of bits. DES is one of the earliest Block Ciphers which was created by IBM, used as a standard for encryption before the 2000's. The requirement for a new encryption standard was needed and AES was designed to supersede DES due to its known vulnerabilities (e.g., full key recovery). In this paper the AES implementation and possible exploitations are explored.**

***Index Terms*—Rijndael, Encryption, Decryption, Secret Key, Modes of Operations, Cryptanalytic attacks, Random number generator, Message authentication, Secure coding, Crypto learning.**

I. Introduction

Currently, Advanced Encryption Standard (AES) is the most used block cipher algorithm. It was chosen by the US government to protect classified information to supersede Data Encryption Standard (DES). On November 26, 2001, this selection was made by the National Institute of Standards and Technology (NIST). AES is a subset of the Rijndael block cipher developed by V. Rijmen and J. Daemen.

AES implementation follows the Federal Information Processing Standards Publications 197 (FIPS PUBS). It can use cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks on 128 bits. The key's length determines the number of rounds to be executed on a given plaintext or ciphertext. AES is a symmetric cryptographic algorithm since the sender and receiver share the same secret key.

According to NIST, there are different recommended modes of operations for the block cipher. They are distinct in several ways such as efficiency, requirements for encryption, and overall security. For example, the most generic mode of operation is Electronic Codebook Mode (ECB). It is efficient because parallel computation is possible, but it is not chosen-plaintext attack (CPA) secure since encryption will always result in the same ciphertext. On the other hand, operation modes implemented for our project, such as Cipher Block Chaining Mode (CBC) and Output Feedback Mode (OFB), are not as efficient but are CPA-secure.

Ideally, a ciphertext would be safe unless the secret key was compromised. AES has been public for several years and different ways to exploit it have been explored. Some of the attacks have been considered to understand the security of the developed project and vulnerabilities that it could still have.

The paper is organized in the following way. Section 2 provides a basic description of the primary methods implemented to achieve AES encryption and decryption. Section 3 provides an overview of various attacks AES or part of AES. It also mentions MAC authentication and key generation. Section 4 goes over the secure coding practices used. Finally, section 5 provides a summary and is followed by references.

II. Implementation

There are three components of AES: Cipher (encryption), Key and Inverse Cipher (decryption). We have followed the pseudo-code given in the standard documentation FIPS 197 [1] for each component. We have focused on two modes of operation in our implementation: Cipher Block Chaining (CBC) and Output Feedback (OFB).

A. Cipher

Responsible for converting plaintext to ciphertext using the Cipher Key through a series of transformations. The following methods are involved: **SubBytes**, **ShiftRows**, **MixColumns**, and **AddRoundKey**. According to the variant of AES, cipher runs for Nr number of rounds.

1) SubBytes: Invertible non-linear substitution operation applied independently on each byte of the state using the S-Box, a 16x16 lookup table. The substitution is determined by the row and column number based on the four most significant bits and the four least significant bits of each byte.

2) ShiftRows: Transform the bytes of the State by shifting them with different offsets. It is defined by the equation below. The first row is not modified.

$$S'_{r,c} = S_{r,(c+shift(r,Nb))modNb} \quad 0 < r < 4, 0 \leq c \leq Nb$$

3) MixColumns: Transforms the state by operating on each column. It treats each value in the column as a characteristic of a finite field with 256 elements, which is also called as Galois Field $GF(2^8)$, multiplies it with a polynomial (obtained from the MixColumns matrix) and takes modulo $x^4 + 1$. The following matrix multiplication can describe this

transformation on the state:

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} x \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}$$

4) AddRoundKey: The AddRoundKey operation performs element-wise XOR of the state with i^{th} word of the round keys generated from the cipher key using the Key Expansion.

B. Key expansion

Our implementation gives two options to users for providing key: provide the secret key path (user-generated key) or securely generate a random key. In the first case, the code verifies the user-given file path for the key and, based on the key length in the file, sets up the environment for encryption or decryption. The second case comes into the picture when the user does not have the key and tells the system to generate one. The system generates a key for the user based on the length requirement and stores it in a hidden directory at a predefined path. We set the Nr, Nk and Nb for all three AES variants according to the provided key length.

AES variant	Key Length (Nk)	Number of Rounds (Nr)	Block Size (Nb)
AES-128	4	10	4
AES-192	6	12	4
AES-256	8	14	4

TABLE I: Key-Round combinations

The key expansion method computes an expanded key of size $Nb * [Nr + 1]$ words from the secret key. Let us briefly mention the operations done in the algorithm that are implemented as methods. In the **Rcon()**, we have used an array of round constants because it is public and known to everyone. It also helps us in faster computation by leveraging some memory. **RotWord()** left-rotates a word by one byte and **SubWord()** is the s-box lookup.

C. Inverse Cipher

In charge of converting ciphertext to plaintext using the Cipher Key through a series of transformations. The following methods are involved: **InvSubBytes**, **InvShiftRows**, **InvMixColumns**, and **AddRoundKey**.

1) InvSubBytes: Invertible non-linear substitution operation applied independently on each byte of the state using the inverse S-Box, 16x16 lookup table. The substitution is determined by the row and column number based on the four most significant bits and the four least significant bits of each byte.

2) InvShiftRows: This method reverses the transformation done by the ShiftRows method. It transforms the bytes of the State by shifting them with different offsets. It is defined by the equation below. The first row is not modified.

$S'_{r,(c+shift(r,Nb))modNb} = S_{r,c} \quad 0 < r < 4, 0 \leq c \leq Nb$

3) InvMixColumns: It computes the reverse transformation of the MixColumns described in the Cipher. The following matrix multiplication describes this transformation on the state.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} x \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix}$$

4) AddRoundKey: The difference between cipher and inverse cipher is that we apply the AddRoundKey in the reverse order. So that it takes values from key expansion for decryption in reverse order of encryption.

D. Modes of operation

*1) **Cipher Block Chaining(CBC)**:* Unpredictable IV is required for encryption in this mode of operation. The same methods used to generate a random key were used to generate IV, which made it unpredictable. The same IV used all over the encryption process has to be used for decryption, and there is no requirement for IV to be secret. As a result, throughout the project, IVs were always the ciphertext's second block, not the standard first block as MAC-tags for authorization purposes were placed there.

Procedures followed by CBC for encryption and decryption are slightly different. Although, the first block will always be initialized with IV values. Afterwards, the previous block will modify the next one. The pseudorandom function (PRF) used, $F_k()$ and $F_k^{-1}()$, are AES's encryption and decryption functions respectively.

Encryption:
$C_1 : F_k(P_1 \oplus IV)$
$C_i : F_k(P_i \oplus C_{i-1}) \quad for\ i = 2, .., n$
Decryption:
$P_1 : F_k^{-1}(C_1) \oplus IV$
$P_i : F_k^{-1}(C_i) \oplus C_{i-1} \quad for\ i = 2, .., n$

Plaintexts that go through CBC encryption must be complete data blocks. They go through the padding for verification. This process adds the required number of extra bits. It starts by determining the number of bytes missing. Once established, bytes with such value are added. For example, if 3 bytes are absent, then the bytes $0x030303$ are attached as the last bytes. If the plaintext already has the required length, an entirely new block will be added to remove padding when decrypting.

*2) **Output Feedback (OFB)**:* This mode of operation also involves an IV but with different requirements than CBC. The IV must be unique but not necessarily unpredictable. As a result, a timestamp was used to generate the IV. There is awareness that an adversary could easily modify a computer's time settings and get the same IV again. There is no need for IV to be secret, so the same format followed in CBC for ciphertexts was implemented for OFB. The first block is a MAC-tag and the second block is the IV.

Plaintexts do not require padding since IV is the only block that goes through F_k for each iteration, and IV is guaranteed to be 16 bytes. The same PRF, F_k, which in this case is AES's encryption, was used for encryption and decryption purposes. Figure 1 shows how OFB is implemented.

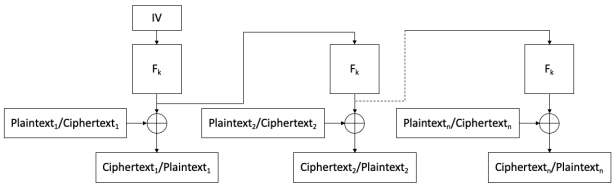

Fig. 1: OFB Encryption and Decryption

III. Crypto Learning

A. CBC-MAC

This is a secure message authentication code that uses a PRF and works for messages of any length. It is widely used in practice and its construction is similar to CBC encryption, although no IV is used. Message's length must to be taken into account when generating the authentication tag for CBC-MAC to be length independent and secure. Figure 2 shows how this works [2]. If used alongside encryption, different keys should be used for authorization and authentication purposes.

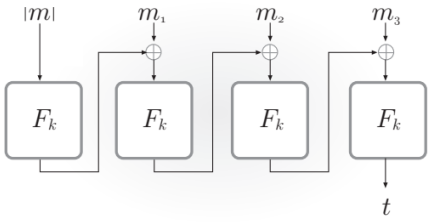

Fig. 2: CBC-MAC of variable length

B. Authentication

Encryption does not provide message integrity since it does not prevent message tampering. [2] Authentication codes are the ones responsible for this task. Without authentication modifying a single bit on a ciphertext would be more than enough for a different plaintext to be obtained when decrypting. The effect would depend on the mode of operation implemented. For example, in CBC, if the ciphertext's IV block is altered, then only the first block of the plaintext would be different. For the rest of the ciphertext, if the i^{th} block was modified, then the i^{th} and $(i+1)^{th}$ blocks of the plaintext would change. On the other hand, in OFB if the ciphertext's IV block were manipulated every block in the plaintext would be modified. For the remaining ciphertext blocks, if the i^{th} block is edited, then the i^{th} plaintext block would change.

Messages encrypted with CBC or OFB are just partially secure if authentication was not implemented. They would be vulnerable to replay attacks, for example. In this attack, honest parties, such as Bob and Alice, exchange an encrypted message; but the message is edited by the adversary. Due to a lack of authentication, Alice has no way to know the message she decrypted is different from the original message Bob sent her.

Therefore, to improve the project's overall security MAC-tags were generated for each ciphertext and appended to them. The encrypt-then-authenticate pattern was followed and the variable-length CBC-MAC was utilized for the MAC-Tags, using different keys for AES and MAC. Decryption was done in the order authenticate-then-decrypt, meaning that a MAC-tag from the given ciphertext was generated and compared to the attached MAC-Tag. If they were equal, then the ciphertext would be decrypted. Otherwise the decryption process would be aborted.

As a result, not only message secrecy but integrity was provided. Replay attacks were prevented, and it also became helpful against other attacks (e.g. oracle padding).

C. Side-channel attacks

Side-channel attacks are a cryptanalytic attack that exploits a system's physical environment to recover some leakage about its secrets. System cache, timing information, power consumption, electro-magnetic (EM) leaks or even sound can provide an additional information source that adversaries can use to exploit the system. [3]

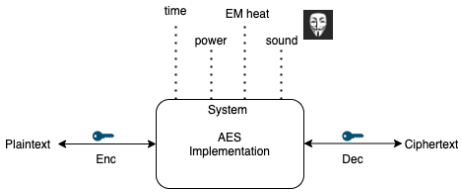

Fig. 3: Side-channel monitoring

Cache-timing attacks are based on attackers' ability to monitor cache accesses made by the victim in the shared physical system as in virtualized environment or type of cloud service. Recently used lines of memory are automatically stored in a limited-size cache, and the RAM cache hits can produce timing characteristics. S-box lookups are table lookups using input dependent indices, and an adversary can leverage this in the subWord phase of AES. For example, if S[0x00] is cached and S[0xFF] is not cached, then reading S[b] takes significantly less time for b=0x00 then b=0xFF leaking information about b.

One of the solutions is to write AES software that takes constant time. That way, it becomes independent of AES input and key and immune to timing attacks. Because table lookups are not constant timed, on-the-fly computation for S-box needs to be done [4]. It uses a total of 128 2-input gates to construct the S-Box: 94 gates are linear operations (XOR and XNOR gates), and 34 gates are nonlinear (AND gates or 1-bit multiplications). However, this approach is much slower than the AES, which uses S-box tables.

Another way to avoid this leak is to guarantee that all AES S-boxes are in cache throughout each AES computation. However, it is not very easy to achieve, and there are issues such as (1) AES S-box lines may be kicked out of cache to make room for memory lines used by computations other than AES. (2) Since caching all S-boxes is very costly, several cache lines can be omitted to make space for the AES computation. (3) Keeping all AES S-box lines in the cache does not guarantee constant-time S-box lookups.

Power analysis attack makes use of power consumption by the hardware during computation. Fluctuation in current generates radio waves, enabling attacks that analyze measurements of electro-magnetic emission. These attacks typically involve similar statistical techniques such as power analysis and called **electro-magnetic attack**.

Because side-channel attacks rely on the relationship between information leaked through a side-channel and the secret data, countermeasures are either to eliminate, reduce the release of such information or to remove the relationship between the leaked information and the secret data (make the leaked information unrelated, or somewhat uncorrelated to the cipher-text). As a solution for these, various techniques such as special shielding to lessen electro-magnetic emissions, power line filtering, insertion of noise or random delays into the emitted channel, can be used.

D. Attacks exploiting properties of MixColumns

There are attacks that exploit the property that each row the matrix used in the MixColumns has two elements of the same value. [5] The idea of the strategy is to choose a set of plaintexts that depend on some guessed bytes of the key and exploiting the fact that some particular property of the state holds for some key with a different property than for the other key. This strategy can be applied in separate differential cryptanalysis attacks to recover the key. In the matrix we used in the MixColumns method, each row has two values equal to 0x01 and thus it is susceptible to such attacks. Though this attack is theoretically possible, it requires a lot of memory and time to exploit the key in a practical approach successfully, and it only works effectively on round reduced variants of AES.

E. Padding Oracle Attack

Padding, used in CBC, brings alongside a strong vulnerability introduced by Serge Vaudenay in 2002 as Oracle Padding. Public-key cryptography standards (PKCS) such as PKCS#5 and PKCS#7 have been found exposed to this attack. It assumes the adversary does not know the secret key but will be able to retrieve the plaintext, given the original ciphertext, by exploiting an oracle that will tell her if there is a padding error on a modified ciphertext. Resulting in a process in which the plaintext is recovered one bit at a time.

For a simple example of how Padding Oracle is exploited, let us assume the adversary, Eve, has access to a CBC ciphertext of two blocks $C = (IV, C_1)$. She does not know how many bytes from C_1 were used for padding, but there is a guarantee that at least one byte. So she can modify IV's value until an padding error is returned, since decryption performs $P_1 = F_k^{-1}(C_1) \oplus IV$. Once the padding error is output, she will know where the plaintext starts and the padding bytes values.

Table II shows what Eve found after modifying the third byte of an 8-byte IV block and getting a padding error, she started by editing the first and second bytes one at a time, but no error was returned. There are still unknown bytes in C_1 (XX) and their value can be found. The padding value found by Eve, $0x06$, could be updated to $0x07$ and a padding error will be returned until C_1's second byte has a value of $0x07$. At that point, Eve will know the value of IV, the output of the CBC decryption for the second byte (0x07), and a way to recover plaintext's original value; since a simple XOR is what is used for CBC.

Index	01	02	03	04	05	06	07	08
Value	XX	XX	0x06	0x06	0x06	0x06	0x06	0x06

TABLE II: 8-Byte Block C_1

The example provided was carried out on a short ciphertext for simplicity, but ciphertexts such as $C = (IV, C_1, C_2, ..., C_T),\ T \geq 1$ could also be exploited through oracle padding. It has been pointed out that padding might not be the only scenario in which an attack like this could be carried out, but whenever the following **general conditions** are met [6]:

1) Formatting rules are checked.
2) An attacker can freely modify an intercepted ciphertext and has access to an oracle.
3) Changes made to the intercepted ciphertext and sent to the oracle are predictable.

In real life, padding oracle has been used to exploit HTTP communication in which MAC authentication was omitted. [7] The project met general condition (1), as formatting was checked for the ciphertext and padding error were thrown to prevent wrong decryption. We are not aware of an adversary having access to an oracle in order to meet general conditions (2) and (3) since symmetric cryptography was implemented and the project executes locally with key access. This did not stop us from implementing authentication through CBC-MAC to prevent oracle padding attacks.

F. Biclique attack on full AES

A biclique attack is a variant of the meet-in-the-middle (MITM) method of cryptanalysis. This attack is slightly better than brute force as it utilizes a biclique structure to extend the number of possibly attacked rounds by the MITM attack.

The biclique helps with the above by allowing us to target seven rounds of AES using MITM attacks and then use a biclique structure covering three rounds of the cipher. It maps the intermediate state at the start of round 7 to the end of the last round, thus targeting the full algorithm. [8]

Biclique attacks on all three variants are somewhat better than the brute-force attack as shown the results from the below table. Even though some advantage is attained, it is just 3 to

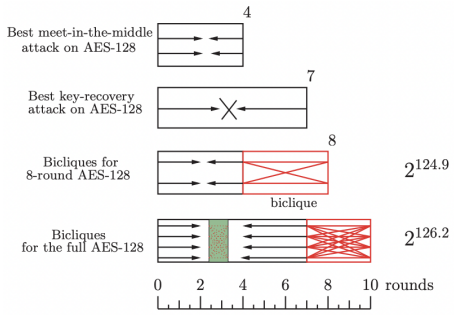

Fig. 4: Biclique attack on AES-128

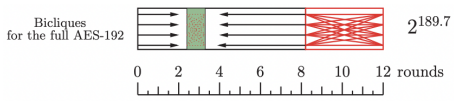

Fig. 5: Biclique attack on AES-192

5 times. [9] This can be considered negligible and therefore, AES is still computationally secure scheme.

rounds	computations	data	memory	success prob.
AES-128 secret key recovery				
10 (full)	$2^{126.18}$	2^{88}	2^{8}	1
AES-192 secret key recovery				
12 (full)	$2^{189.74}$	2^{80}	2^{8}	1
AES-256 secret key recovery				
14 (full)	$2^{254.42}$	2^{40}	2^{8}	1

TABLE III: Biclique Key Recovery for AES

G. Ciphertext stealing

We have explored the following countermeasure which can be implemented to avoid padding oracle attack on CBC Mode of operation. [10] Padding is used to make the length of plaintext a multiple of block size. It might result in a vulnerability which Padding Oracle attacks can exploit. This can be overcome with the ciphertext stealing technique. In that, a portion of the second last block of ciphertext is taken and appended to the last block of plaintext and this padded message is then encrypted to generate the final block of ciphertext. The ciphertext portion that was appended to plaintext in the final block is discarded from the ciphertext. During decryption, the final block is decrypted first, and the output is used to restore the ciphertext of the previous block which is then decrypted as usual.

H. Key Generation

Key is the most crucial part for preserving the secrecy of the message as we have studied in Kerckhoffs's principle, "Cryptographic security must rely on a secret key and not on the algorithm." [2] AES follows this notion as it should, and therefore, the user needs to wary of correctly generating the key and safeguarding it. A key has to be random enough that it becomes unpredictable. If we are using this to seed a pseudorandom number generator (PRNG), it should be a PRF.

- We studied and discarded many approaches to generate random number because of their limitations and ineffectiveness. E.g. standard library functions such as rand() and srand(), Mersenne Twister PRNG, online shared data on claimed random resources, predictable entities like time, PIDs and temperature and not designing our own PRNGs.
- Information gathered from various sources; we concluded that using a non-deterministic random number generator (NDRNG) that produces the random numbers from an entropy pool maintained by the underlying Operating System is considered the best way to get a random number.
- The Linux kernel generates the entropy from keyboard timings, mouse movements, and IDE timings and makes the random character data available to other operating system processes through the particular files /dev/random and /dev/urandom.
- Both /dev/random and /dev/urandom are fed by the same cryptographically secure PRNG (CSPRNG). The behavior differs (according to some estimate) only when their respective pool runs out of entropy. In that case, /dev/random blocks, while /dev/urandom does not.
- Figure 6 shows how Linux kernel generates random numbers.
- When using /dev/random it may come across a scenario where Kernel has to wait for cycles and lose precious time. We decided not to use /dev/random because of its properties like unavailability, unusability, and inconvenience.
- /dev/urandom is a result of the CSPRNG based on "the current level of entropy" in the system. That is why we decided to use it as the randomness source for our project. The only instance where /dev/urandom might imply a security issue due to low entropy would be at fresh bootup of OS or install.
- Coming towards the implementation, we have used C++ wrapper function std::`random_device`, which opens the system's /dev/urandom file and returns 32 bits unsigned integer. We make an appropriate number of system calls based on key-size Nk and get a random key for the user.

Another option is to open the file /dev/urandom explicitly and get key using fopen("/dev/urandom"). An enhancement for this system is to seed the NIST approved deterministic random bit generator (DRBG) or other trusted PRNG with the above results and use generated key from that. [11]

- Hash-based DBRG mechanisms (Hash-DRBG or HMAC-DRBG). Underlying algorithms are SHA family hash functions (SHA-1, SHA-256/512, etc.)
- A system with support for both hash functions and block ciphers might use the CTR-DRBG if the ability to parallelize the generation of random bits is needed.

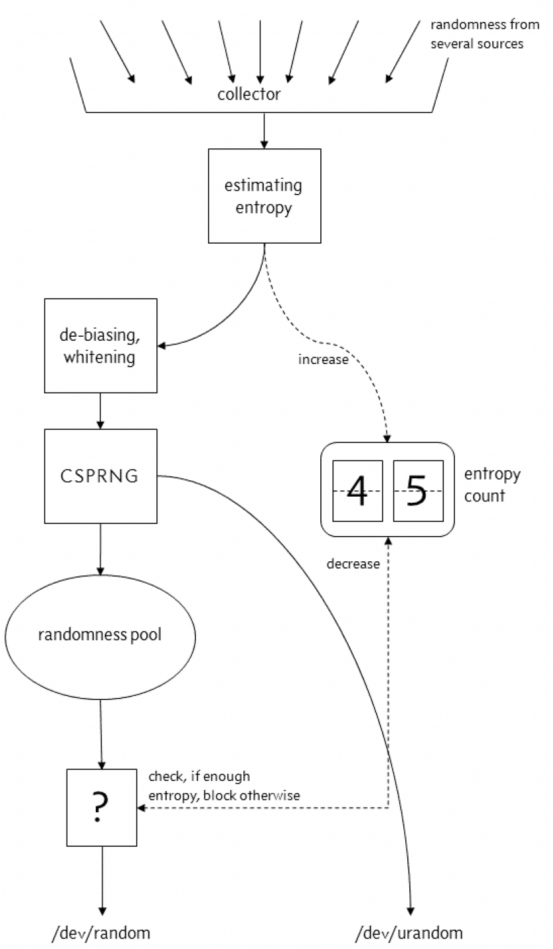

Fig. 6: Structure of Linux kernel's random number generator

CTR-DRBG uses an approved block cipher algorithm in the counter mode (TDEA, AES).

- An AES-based DRBG mechanism (i.e., CTR-DRBG using AES) itself is the best choice for us because we already trust AES.

This way we are obtaining a random key for the user.

I. Key Storage

In symmetric cryptography, a key is shared between the sender and the receiver but kept secret from the intruder. If an adversary has access to the secret key, the security of any ciphertext that was encrypted with it would be compromised. Therefore, proper key storage was considered throughout the project. Database implementation was discarded as it would bring new risks outside the cryptography scope (e.g., SQL-Injection, Weak Authentication, DoS).

As a result, secret keys used for authorization and authentication are stored locally by default. This was mainly done to facilitate execution and testing. Small contributions such as storing them in hidden folders as hidden files were implemented. Most importantly, the user can also store keys directly when generated on a USB drive. It is not a perfect scenario, as a compromised USB storage could still have physical unauthorized access; but USB drives empower symmetric key distribution. [12]

J. Key Zeroization

Whenever program execution is finished, it deallocates the memory used but does not clean up the populated value by the program. Therefore, we have introduced memory zeroization to make our system backtracing resistant. All keys and intermediate states computed, while plaintext encryption and ciphertext decryption are meant to be ephemeral. They are destroyed when released by the appropriate methods. The application is responsible for calling the destruction function that overwrites the memory occupied by confidential data with "zeros" and (wherever necessary) deallocates the memory.

In the case of a failure, we throw an exception to let the user know about it. An enhancement of the key zeroization step could be like this, which we have not implemented: in case of abnormal termination, or swap in/out of a physical memory page of a process, the keys in physical memory are overwritten by the Linux kernel before the physical memory is allocated to another process.

IV. Secure Coding

The Software Engineering Institute (SEI) has established C++ coding standards. These rules reflect the current thinking of secure coding, and we have followed them as much as possible in AES implementation. Examples of how the requirements were satisfied or not are provided with the rules.

File I/O are sensitive operations and must be handled carefully. Closing files at the end of their scope is of the same importance as not accessing closed files. When a file is opened with update mode, performing both read and write on the same stream may lead to undefined behavior and must be avoided. We have followed all of these in our methods.

- **FIO50-CPP**: Do not alternately input and output from a file stream without an intervening positioning call.
- **FIO51-CPP**: Close files when they are no longer needed.
- **FIO46-C**: Do not access a closed file.

We have followed these expression-based rules that focus on taking care of the ambiguity in code (i.e., a[i++] = i), not accessing out-of-scope memory variables or objects, and treat bitwise and boolean operations differently by using appropriate operators for both of them.

- **EXP50-CPP**: Do not depend on the order of evaluation for side effects. Throughout the project, side effects were taken into account, avoided in general for proper execution.
- **EXP53-CPP**: Do not read uninitialized memory. Constructors helped us avoid this by adequately initializing variables before they were used. Variables outside a constructor's scope were individually initialized before dealing with them.
- **EXP54-CPP**: Do not access an object outside of its lifetime.

- **EXP46-C**: Do not use a bitwise operator with a Boolean-like operand. To meet this rule's requirements, logical operators (e.g. &&, ||) were used.

The range-based scenarios such as loops are handled securely by considering these rules.

- **CTR53-CPP**: Use valid iterator ranges. This was achieved by the use of a range-based for loops when iterating through a vector.
- **CTR52-CPP**: Guarantee that library functions do not overflow. Vectors were used in certain scenarios, such as blocks, to avoid overflow. Arrays were still utilized in States, and arrays sizes were verified when initialized or edited.
- **ARR30-C**: Do not form or use out-of-bounds pointers or array subscripts. As an example of how we achieved this, the Sequence class had an array and an int to keep track of the size of its array. Whenever a Sequence's array was iterated it was according to its array's size, and if a Sequence array changed so did its size.

When declaring and initializing variables, we have taken the given practices into account.

- **DCL52-CPP:** Never qualify a reference type with const or volatile. References were properly stated, for example: const Block& block.
- **DCL60-CPP:** Obey the one-definition rule.

Proper allocation and deallocation of memory is a crucial part of memory management. We developed destructors for this purpose besides clearing sensitive information such as key expansion and messages.

- **MEM50-CPP:** Do not access freed memory.
- **MEM51-CPP:** Properly deallocate dynamically allocated resources.
- **MEM53-CPP:** Explicitly construct and destruct objects when manually managing object lifetime. We achieved this by having constructors and destructors for each class. Therefore, having manual control and manageability of an object's lifetime.
- **MEM52-CPP:** Detect and handle memory allocation errors.

The code can handle as many thrown exceptions from the critical parts, such as' padding errors'. However, exceptions may arise in a wide range for many scenarios, and there are certainly a few we did not consider.

- **ERR51-CPP:** Handle all exceptions.

We have followed the following coding practices for cryptography which cover: operations, certain environment scenarios and more.

- **MSC52-CPP:** Value-returning functions must return a value from all exit paths. Non void functions always return a value in order to proceed with proper execution.
- **MSC41-C:** Never hard code sensitive information. In our code, an adversary could easily find where the keys are stored by the program, the hidden '.crypto' directory, and the default name given to the keys files by the program. This was hardcoded to facilitate execution, and it is not the only option has to store the keys, but it could be compromising.
- **ENV30-C:** Do not modify the object referenced by the return value of certain functions. It is common practice to state get() methods as const as they do not modify the value. Unfortunately, having some get() methods with this condition resulted in errors as the returned value was modified. To facilitate the process, we avoided the const clauses, and therefore avoided this rule.
- **INT33-C:** Ensure that division and remainder operations do not result in divide-by-zero errors. These two operations were used as part of the subBytes, subBytes, and invSubBytes process and the requirements were met.

V. Summary

The project has been a great learning experience from both a coding and conceptual point of view. Started with the implantation of AES in C++, soon it became a multifaceted project which enhanced our knowledge greatly on many aspects and use-cases of C++ (DS, bit-wise operations, file IO, libraries), conceptual knowledge of Rijndael block cipher (Galois field, modes of operations, MAC) and system knowledge (RNG). As we dived deep to learn more about the attack, we understood the vulnerabilities of the system (side channel, cache) and algorithm (biclique) or a part of the algorithm (oracle padding). We learned about the intricacies of secure coding and crypto during the implementation. As challenging and demanding the project was, the pressure was lessened by our teamwork, and we must admit that the final result was satisfactory.

References

[1] (2001) Advanced encryption standard (aes). [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[2] J. Katz and Y. Lindell. (2007) Introduction to modern cryptography: Principles and protocols.

[3] D. J. Bernstein. (2005) Cache-timing attacks on aes.

[4] A. Ghoshal and T. D. Cnudde. (2017) Several masked implementations of the boyar-peralta aes s-box. [Online]. Available: https://eprint.iacr.org/2017/1023.pdf

[5] L. Grassi. (2017) Mixcolumns properties and attacks on (round-reduced) AES with a single secret S-Box. [Online]. Available: https://eprint.iacr.org/2017/1200.pdf

[6] V. Klíma and T. Rosa. (2003) Side channel attacks on cbc encrypted messages in the pkcs7 format. [Online]. Available: http://spi.unob.cz/papers/2003/2003-11.pdf

[7] J. Rizzo and T. Duong. (2010) Practical padding oracle attacks. [Online]. Available: https://static.usenix.org/event/woot10/tech/full_papers/Rizzo.pdf

[8] A. Bogdanov, D. Khovratovich, and C. Rechberger. (2011) Biclique cryptanalysis of the full aes. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-25385-0_19

[9] A. Bogdanov, C. Rechberger, and D. Khovratovich. (2005) Biclique cryptanalysis results. [Online]. Available: https://www.iacr.org/archive/asiacrypt2011/70730339/70730339.pdf

[10] M. Dworkin. (2010) Recommendation for block cipher modes of operation: Three variants of ciphertext stealing for cbc mode. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a-add.pdf

[11] E. Barker and J. Kelsey. (2015) Recommendation for random number generation using deterministic random bit generators. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final

[12] R. Kumaravelu and N.Kasthuri. (2010) Distribution of shared key (secret key) using usb dongle based identity approach for authenticated access in mobile agent security. [Online]. Available: https://ieeexplore.ieee.org/document/5738790

APPENDIX

A. C++ Code

Please find the code at https://github.com/nkuntal/AES/tree/main/AES

B. Crypto Practices Learned

- **MSC50-CPP:** Do not use std::rand() for generating pseudorandom numbers.
- **MSC51-CPP:** Ensure your random number generator is properly seeded.
- Encryption does not provide message integrity, because it does not prevent message tampering. [2]
- **Compare secret strings in constant time.** String comparisons performed byte-per-byte may be exploited in timing attacks, for example, to forge MACs.
- **Avoid branching controlled by secret data.** If a conditional branching (if, switch, while, for) depends on secret data, then the code executed as well as its execution time depends on the secret data as well.
- **Avoid table look-ups indexed by secret data**. The access time of a table element can vary with its index (depending on whether a cache-miss has occurred). Cache-timing attacks on AES is such an example. One possible solution is to replace table look-up with sequences of constant-time logical operations. S-box could be implemented this way.
- **Prevent compiler interference with security-critical operations**: It may hinder memory zeroization (i.e. memset() method), leading to data leak if not done correctly.
- Use unsigned bytes to represent binary data.
- **Default Deny**: The system should grant base access decisions on permission rather than exclusion. Therefore, by default, access is denied and the protection scheme identifies conditions under which access is permitted.

C. Secure Coding Practices Learned

1) Used:

- **FIO50-CPP:** Do not alternately input and output from a file stream without an intervening positioning call.
- **FIO46-C:** Do not access a closed file.
- **FIO51-CPP:** Close files when they are no longer needed.
- **EXP46-C:** Do not use a bitwise operator with a Boolean-like operand.
- **CTR53-CPP:** Use valid iterator ranges.
- **DCL52-CPP:** Never qualify a reference type with const or volatile.
- **EXP53-CPP:** Do not read uninitialized memory.
- **EXP54-CPP:** Do not access an object outside of its lifetime.
- **EXP50-CPP:** Do not depend on the order of evaluation for side effects.
- **CTR52-CPP:** Guarantee that library functions do not overflow.
- **MSC52-CPP:** Value-returning functions must return a value from all exit paths.
- **MEM50-CPP:** Do not access freed memory.
- **MEM51-CPP:** Properly deallocate dynamically allocated resources.
- **MEM53-CPP:** Explicitly construct and destruct objects when manually managing object lifetime.
- **ARR30-C:** Do not form or use out-of-bounds pointers or array subscripts.
- **INT33-C:** Ensure that division and remainder operations do not result in divide-by-zero errors.

2) Not Used:

- **ERR53-CPP:** Do not reference base classes or class data members in a constructor or destructor function-try-block handler.
- **ENV30-C:** Do not modify the object referenced by the return value of certain functions.
- **MSC41-C:** Never hard code sensitive information.
- **ERR51-CPP:** Handle all exceptions.
- **MEM52-CPP:** Detect and handle memory allocation errors.
- Adhere to the **principle of least privilege**. Every process should execute with the least set of privileges necessary to complete the job. Any elevated permission should only be accessed for the least amount of time required to complete the privileged task. This approach reduces the opportunities an attacker has to execute arbitrary code with elevated privileges.