An Investigation on Implementations of Theoretical Whitebox Cryptographic Solutions

by

Adnan Rahman Eshan
20101601
Jarin Tasnim Khan Kashfee
20101062
Md.Rabib Hasan
20101561
Mahmudul Islam
20101200

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
January 2024

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.

2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.

3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.

4. I/We have acknowledged all main sources of help.

Student's Full Name & Signature:

Rabib Hasan	[illegible]
Md.Rabib Hasan 20101561	Jarin Tasnim Khan Kashfee 20101062
Mahmud.	[illegible]
Mahmudul Islam 20101200	Adnan Rahman Eshan 20101601

Approval

The thesis titled "An investigation on implementations of Theoretical Whitebox Cryptographic Solutions" submitted by

1. Mahmudul Islam (20101200)
2. Md.Rabib Hasan (20101561)
3. Jarin Tasnim Khan Kashfee (20101062)
4. Adnan Rahman Eshan (20101601)

Of Fall, 2023 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on Jan 22, 2024.

Examining Committee:

Supervisor:
(Member)

Muhammad Nur Yanhaona

Muhammad Nur Yanhaona
Associate Professor
Department of Computer Science and Engineering
Brac University

Program Coordinator:
(Member)

Md.Golam Rabiul Alam, PhD
Associate Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi
Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Abstract

Whitebox Cryptography techniques are those which are aimed at protecting software implementations of cryptographic algorithms against key recovery in unprotected devices. The sensitive data embedded in the code is the major concern in any security-sensitive application. Therefore, data encryption is indispensable. But white box cryptography aims to protect the security keys used for data encryption from being revealed. As a whole, such a type of cryptography concerns the analysis of algorithms that are said to operate in a whitebox attack context. In this attack context, all information and details of implementations are visible to an attacker. The attacker not only sees the input and output portions, but they can also see every intermediary implementation and operation that happened along the way. The challenge that whitebox cryptography aims to address is implementing a cryptographic algorithm that will keep the cryptographic assets of software secure even when subject to whitebox attacks. As converting blackbox cryptographic algorithms to whitebox has some sort of performance consequences that have not been measured or identified properly, we will compare the performance of alternative algorithms. In addition to that, as some of the algorithms have very few implementations and any effective open source implementation is yet to be found that will help the researchers in this sector, we will write an open source white box implementation for our performance analysis and the future benefit of the researchers. Moreover, companies lacking resources /skills may also be benefited because of such implementation. In addition, we want to investigate the kinds of attacks that can be launched against the solution in our library and try to add security features to strengthen them to resist common forms of attacks.

Table of Contents

Chapter 1

Introduction

The defense mechanism of cryptographic algorithms used in untrusted contexts is an important concern in a time when digital interactions and outsourced computations are in common. There are four principles of modern day cryptography. They are data integrity, data confidentiality, authentication, and non-reputation. The initial goal of cryptography was to save the integrity and confidentiality of data from eavesdroppers. This goal has resulted in lots of ciphers and algorithms such as AES, DES, RSA, and ECC. In the implementation of these algorithms, the endpoint is trusted which gives recognition to a type of cryptography known as Black Box Cryptography. In this type of cryptography, the user has only access to the input and output of the algorithm. The Black Box context which is the conventional one is like a real Black Box where the attackers can only see the output. While cryptographic approaches were first developed with this environment in mind, symmetric ciphers in particular are frequently tested for BBC errors. But now the environment has changed and it is leading to situations like Digital Rights Management (DRM), where users want to gain access to protected information outside of the constraints and restrictions set by DRM software and this software operates in environments that can be possibly harmful. Eventually, the problem worsened when cryptography was deployed in open devices such as PCs, tablets, and smartphones without exploiting security issues which gave recognition to a new type of attack context known as white box attack context where the attacker has full access to the software implementation of the algorithm. In another way, we can say that the attacker can see the binary and alter it. When this incident happens the conventional model Black Box becomes ineffective and then we use White Box Cryptography to protect the digital assets.

In short, it can be said that the implementation is the core line of defense here. Software implementations that are capable of handling these sorts of attacks are called White Box implementations. The main idea of White Box implementation was to embed the fixed data and random data in a composition from which it's hard to derive the original key. As a whole, in White Box AES implementation, the technique transforms the cipher into a lookup table. The secret key is hardcoded into the lookup table and protected by randomized techniques that are applied. In the Black Box, the key is public but in the White Box Cryptography, the key is embedded. In short, White Box Cryptography is a specialized field dedicated to protecting cryptographic algorithms from the challenges that are presented by untrusted execution contexts which is a domain known as the Whitebox context (WBC) and this is an important part of the study of the subject.

The study of White Box Cryptography is related to the attacker model developed by Chow et al. in 2002. Within the WBC, an effective set of tools is available to attackers, and with these tools, they can closely follow the algorithm's precise flow, analyze memory use, and access instructions during calculations. They can perform any kind of operation like - changing cycle counters, conditional execution (down to a single round of a cipher), conditional statements, program memory, the real-time execution environment, and even fault induction.

However, the White Box DES implementation was first shown as insecure due to its vulnerability in the DES Feistel structure which can be distinguished in a lookup table representation. In the same manner, the AES implementation was broken using an algebraic technique. After that many constructions were proposed but the security of these implementations remains unclear. Moreover, their implementation is not available in an open-source format because they use it for their security purpose.

The two main attacks that rule the field of White Box Cryptography are Plaintext Recovery under Chosen Plaintext Attack (PR-CPA), which enables decryption using an implementation of the cipher with an embedded encryption capability. Another one is Key Recovery, which demands the extraction of an embedded symmetric key. This effort is connected to the White Box cryptography project, a software obfuscation-related effort where the technique uses unique security concepts, most importantly cipher invertibility (PR-CPA), to go beyond obfuscation. White Box Cryptography intends to implement cryptographic algorithms in software in a way that protects cryptographic assets even when they are exposed to White Box attacks. These software implementations which are known as White Box implementations exhibit the strength needed in the modern computer environment to survive adversary testing. As a whole, the main idea of White Box implementation was to embed the fixed data and random data in a composition from which it's hard to derive the original key.

This thesis work started with a required thorough investigation of different modes of the Black Box AES algorithm and the implementation and performance analysis of the conversion of the Black Box cryptographic algorithm to the White Box cryptographic algorithm. Moreover, this thesis investigates the use of the Advanced Encryption Standard (AES) cipher in the context of both Black Box Cryptography and White Box cryptography and we believe this area of White Box cryptography is still in its early stages. Moreover, their implementation of White Box is not available in an open-source format because they use it for security purposes. So by getting acquainted with various white box implementations and how they are broken using cryptanalysis techniques, side-channel attacks, and other attacks the efficiency of the White Box implementation can be proven to the fullest level. That is why, our journey through this thesis includes a series of analyses of different modes of Black Box encryption and an open source implementation of the Conversion of Black Box cryptographic algorithm to White Box cryptographic algorithm including the challenges it encounters, the transformation techniques it uses, and the constant search for improved defense against adversarial attacks. The research of White Box cryptography demonstrates the cryptographic community's ambition to adapt and strengthen the fundamentals of secure digital communication as attackers become more persistent and digital trust boundaries continue to shift. This thesis emphasizes the significance of ongoing research in addressing evolving security challenges in today's interconnected world. It provides insightful observations, in-depth analysis, and potential procedures for enhancing security in the modern computer environment[3].

1 Problem Statement

Whitebox cryptography is designed to defend against key recovery for software that implements cryptographic algorithms. An attacker can see all information and implementation details in the context of this attack. Implementing a cryptographic algorithm that will keep a software's cryptographic assets secure even when subject to white box attacks is the problem that Whitebox cryptography seeks to answer. As a whole, implementing the white box implementation is the sole line of defense in this sector. Moreover, the implementation of an open-source library is yet to be found which will help to measure the performance consequences of the algorithms which are discussed as solutions to the problem.

In addition, the algorithms that are there to convert from Black Box cryptography to Whitebox cryptography have performance consequences as there are no open source libraries to investigate the security and performance characteristics of the algorithms. Furthermore, commercial products are so costly that the smallest and medium software firms in Bangladesh cannot purchase them.

On the whole, implementing solutions and constantly investigating their security and performance characteristics using different tools and side channel attacks can eventually help to set up an open source verified Whitebox implementation.

2 Research Objectives

Throughout the research, we seek to protect the security key from being revealed. The objective we aim to achieve is as follows:

1. Measuring the performances of alternative algorithms of converting from Black Box cryptography to Whitebox cryptography.

2. Writing an open source Whitebox implementation for performance analysis.

3. Investigating the kind of attacks that can be launched against the solution in our library and if possible trying to strengthen the solution.

Chapter 2

Literature Review

2.1 Fundamentals of Cryptography

Communication is done between the sender and the receiver but in between, there is an eavesdropper too. Eavesdropper is harmful to the system so to overcome the problem an approach is used named public key distribution [2]. Here in this approach though the eavesdropper can hear the message passed between the sender and receiver but he is unable to decode the key so cannot understand the conversation between the sender and receiver. There is another approach to overcome the problem called public key cryptosystem [2]. This approach allows the creation of separate keys that can be used for enciphering and deciphering. Enciphering and deciphering are two important functions. If they are separated then there will be privacy as if access is given to encipher a message, without any doubt it is possible to decipher the message too so they both are to be protected using separate keys. Some cryptographic techniques are discussed below-

In substitution, the key is a permuted alphabet so the plain text is every time replaced with its following permuted alphabet. This technique is generally known to puzzle resolvers. Next, there is a transposition, here the message is broken down into character groups and the letters of each group are organized based on permutation, and then on the specific position number they are being written. Next, there are polyalphabetic ciphers. To encipher messages, multiple substitution alphabets are used from period to period. Such as – (Alphabet * n + i) where n is the number of periods, and i is the ith alphabet to be enciphered. Next, there is the running key cipher which is aperiodic polyalphabetic Here the key is the message and text that are taken from a book and then the plain text is replaced with the key.

In Cryptography, the most used application is the Galois field [12]. Here bytes are constituted as vectors and with the help of arithmetic encryption and decryption are done smoothly. But before data encryption is done, data is to be organized in a matrix of bytes. There is an algorithm for AES that contains SubBytes, ShiftRows, MixColumns, and AddRoundKey [12]. With the help of the algorithm, it is possible to encrypt data, and by decrypting data the algorithm can be applied backward. Data Encryption Standard (DES) was developed in the early 70s by IBM where a 56-bit key is used by DES and a supercomputer could break the key in just a day or less. So, a polished algorithm was important. Hence, the Rijndael algorithm came on lead and from, and since then it has Advanced Encryption Standard or AES [12].

In [5] it can be seen that in Rijndael, many operations are performed in bytes. An array or a matrix containing 4 rows and 4 columns is created and in here each entry is a byte or 8 bits such a way that they are a total of 16 bytes. The Rijndael cipher consists of three rounds – initial RoundKey addition round, Nr -1 rounds, and lastly final round. Implementation of Rijndael in 8-bit and 32-bit processors is discussed in [5]. Limitations of inverse with its cipher are taken under consideration as the cipher is barely suitable for use on the smart card itself. In software, both the cipher and its inverse have their tables which are different from each other.

In modern security applications, secret key cryptography and public key cryptography are the major cryptographic systems that are used. Secret key cryptography is also known as symmetric key cryptography and it uses a single key for both encryption and decryption. The key needs to be shared and kept secret in the communication process. Examples of secret key cryptography are AES, DES, etc. Public key cryptography is also known as asymmetric key cryptography and it uses two mathematically related keys - a private key and a public key. Examples of public key cryptography are RSA, ECC, etc. The public key can be openly shared and it is used to encrypt messages. The private key is kept secret which is used to decrypt messages. The choice of cryptography depends on the specific needs of the user as both have their weaknesses and strengths. The main focus of our thesis is on secret key cryptography, in particular AES.

2.2 Blackbox Cryptography Overview

The Blackbox restricts access to viewing the internal actions of software. As the main purpose of an attacker is to extract the key for implementation, Blackbox obfuscation is nearly impossible to decode. So the working behavior of the blackbox goes from observing the inputs and outputs, checking the text of the input, and restricting access to the entire execution process. Among the attacks of the Blackbox, there are 3 progressive levels. They are discussed below - *Passive attacks* which are nothing but plaintext attacks that are known and can view the inputs and the output in the system. Next *active attacks* are the plaintext attacks that are chosen and they can imply interaction. Lastly, *adaptive attacks* which are plaintext-cipher text attacks that are dependent on the results of the previous interchange [4].

Blackbox attacks come in progressively more sophisticated forms in the realm of cryptography. Advanced attacks take advantage of the knowledge of an algorithm's internal details while appearing to be 'black boxes' at the time of execution. Users of cryptography are vulnerable to a variety of risks and assaults that are undetectable in a blackbox setting [6].

Blackbox attack models are way too idealistic for software applications on unreliable hosts. Typical smartcard defenses are ineffective because of the WBC's assumption that the attacker has unrestricted access to the implementation. According to the SETUP (Secretly Embedded Trapdoor with Universal Protection) mechanism an attacker can steal the user's secret covertly while guarding against outside attacks and reverse engineering. Cryptosystems can be changed to leak key information with a greatly reduced danger of being detected [6].

2.3 Whitebox Cryptography Overview

The primary concept of Whitebox cryptography is to integrate the random data which is initiated at compilation time and the fixed key in the form of data but also the form of code in a composition which will create difficulties in deducing the original key [23]. In the Whitebox, the attacker can view the process as the algorithm is known to the attacker and after viewing the internal actions of a system, can control the text of the inputs. This also gives the attacker memory access. After the application of Whitebox Cryptography, the key cannot be extracted as the key is not kept in the memory.

Code obfuscation is related to Whitebox cryptography. Their common goal is to protect software implementation. Even though they both use the same theoretical framework, the security of the latter needs to be validated concerning security notions. Additionally, Whitebox cryptography's theoretical study demonstrated that it serves as a link between symmetric and asymmetric encryption [23]. Although a new type of public scheme can be formulated where a symmetric key is used effectively by a block cipher to perform a private operation, the same symmetric key can be used for public operation in white box implementation. In the case of implementing Whitebox encryption of AES, WB-AES is converted to AES into a set of look-up tables and hiding the secret keys within the table [1]. In contrast, the Whitebox attack context considers threats that are much more serious. Firstly, it is predicted that the fully privileged attack software shares access to algorithm implementation on the same host as cryptographic software. Secondly, with the help of instantiated cryptographic keys, dynamic execution can be observed. Lastly, internal algorithm specifications are fully observable and freely editable [4]. Whitebox Cryptography is found to be more suitable than Blackbox Cryptography as the algorithm is protected thus it makes the whitebox more secure and suitable and restricts threats.

Implementation in the Whitebox Cryptography -

WB-DES implementations: It was first proposed in the year 2002 but it was broken by a fault attack in the same year. In the year 2005, the implementation was improved but it was also broken by differential cryptanalysis in the year 2007.

WB-AES implementations: The first approach was proposed in the year 2002 but it was broken by the BGE attack in the year 2004. The latest attack in the year 2013 was able to extract keys from an improvement with 2^{22} complexity.

Public Whitebox Cryptography implementations -

1. *Wyseur WB-DES challenge* - The very first publicly available whitebox challenge was created by Brecht Wyseur in the year 2007 [11][16] The challenge was solved independently in the year 2012 by James Muir and "SysK" using differential cryptanalysis[13].

2. *Hack.lu 2009 WB-AES challenge* - The challenge was solved by Eloi Vanderbeken, he reverted the functionality of the white box implementations from encryption to

decryption and it was also solved by “SysK”, he managed to extract the secret key from the implementations[13] [22]

3. *SSTIC 2012 WB-DES challenge* - The challenge was also solved by five participants, it was an implementation that used no encodings[20].

4. *Klinec WB-AES based on dual ciphers* - The attack proposed recovers the key automatically[1].

5. *The NoSuchCon 2013 WB-AES challenge* - The challenge was completed by a number of participants but without recovering the key. It used external encodings so it was not vulnerable to the proposed attack[16][24].

2.4 Attacks on Whitebox Cryptography

2.4.1 Cryptanalysis Attacks

In [8] it states about the BGE attack, an effective attack in the case of AES implementation. In the discussion, a detailed explanation is stated to remove the secret key of AES with minor memory requirements and worse time complexity. The main purpose of the attack is if the lookup tables are considered then there could be the possibility of leaking sensitive information as through three consecutive encoded round analyses the attacker can recover the 128-bit AES secret key easily. In [1] it states that the improvement of the BGE attack, about how time-consuming it was before and then the work factor of the BGE attack, is reduced to 2^{22} after the implementation. In the BGE attack, the AES key can be extracted with the work factor of 2^{30}. Later on, Tolhuizen's improvement was done on the BGE attack and the work factor of the BGE attack was reduced to 2^{22} after implementation. Later on, a new attack was introduced which was on the implementation of the initial Chow et al [4]. This attack was on the output bytes on the first encoded AES round. Here in this new attack, the work factor was also found to be 2^{22}. The karroumi's white box implementation consists of two phases namely dual AES cipher and techniques of Chow et al and this implementation was done to repel the BGE attack. Later on, the BGE attack of Chow et al and Karroumi was found similar. So, both attacks on [14] could be used to extract the key from Karroumi's Whitebox implementation.

2.4.2 Fault-Based Attacks and Protocols

In [3] it discusses the attack that creates the usage of hardware faults about how using hardware faults, cryptographic/ encryption schemes can be broken. Attacks have been made on the implementations of the Chinese remainder theorem (i.e. e RSA and Rabin) as they are sensitive to the attack. Then it is found that using hardware faults to attack the protocols could also be applicable to authentication schemes such as Fiat – Shamir and Schnoor identification protocols. The two protocols could also be broken using hardware faults. Similarly, other identification schemes are also applicable for such.

2.4.3 Differential Fault Analysis (DFA)

Based on algebraic properties of modular arithmetic, a type of attack was introduced that only applies to public key cryptosystems (i.e. – RSA) but later on a new attack was introduced named Differential Fault Analysis (DFA) [9] which can apply to any kind of key cryptosystem. This attack can be useful for recovering any cryptographic hidden secrets in the tamper-resistant device. Later on, it is found that to identify the keys of unknown ciphers in the tamper-resistant devices, techniques are processed. After that, a faulty model is taken under consideration which has permanent hardware faults, and then it is shown how it can be used to break DES. [7] JeanGreyis a set of tools that can perform differential fault analysis (DFA). PhoenixAES is a tool to perform differential fault analysis attacks against AES.

2.4.4 Differential Computation Analysis (DCA)

Differential Computation Analysis is a software-based attack that is performed on Whitebox Cryptography implementations[17]. It involves tracing a program's execution and recording the memory addresses during execution. It generally uses tools like Pin and Valgrind. It identifies a cryptographic algorithm that is based on visuals and records multiple traces with different inputs. For our implementation, we have used Valgrind to perform our Whitebox cryptography implementations.

2.4.5 Side-Channel Attacks

A survey was conducted on side-channel attacks [11] [21] about how sensitive and private the attacker can recover information and ways to overcome the problem. Here in side channel cryptanalysis, the information is preserved in the form of a physical implementation of a cryptosystem. This attack doesn't attack with the help of an algorithm but it attacks with the physical implementation. Among the side-channel attacks, there are – Timing attacks, Power analysis attacks, Electromagnetic analysis attacks, fault induction attacks, Optical side-channel attacks, and traffic analysis [23].

Differential Power Analysis :

DPA is a side-channel attack that is applied to hardware cryptosystems[17]. It involves analyzing power consumption traces or hardware power traces statistically during cryptographic operations. It targets to identify the key-dependent power consumption patterns that will reveal secret keys. The main target of an attacker for recovering the part of the key k is they compare the real power measurements t_i of the device with an estimated power consumption under all possible hypotheses for key k.

Chapter 3

Implementation of Blackbox AES

In every implementation of cryptographic algorithms, Galois field algebra has intrinsic importance. The paper entitled "Galois Field in Cryptography" by Christoforus Juan Benvenuto [12] gives a brief description of the operations of Galois field algebra. Galois field is particularly useful in the case of computer data translation as computers only understand binary data. Essentially, the 0s and 1s that make up computer data are stored in a two-element Galois field and after that, it is feasible to carry out mathematical operations that efficiently and effectively modify and scramble the data by treating the data as vectors in this field.

3.1 Galois Field Algebra in Cryptography

Galois Field are essential because they allow data that has to be computed to be represented as vectors within a finite field, simplifying and streamlining mathematical procedures. Along with the application of the Galois field in different cryptographic algorithms, the paper [12] also presents a historical context of cryptographic algorithms. At the very beginning, the historical perspectives included the Data Encryption Standard (DES), which employed a relatively small 56-bit key and was first introduced by IBM in the 1970s. However, as technology developed, DES became increasingly open to attacks, necessitating the development of stronger encryption algorithms. In response, Vincent Rijmen and John Daemon created the Rijndael algorithm in 2001. As a result of its improved security characteristics, AES later gained widespread adoption. The usage of the Galois field in AES algorithm is given here-

1. While doing addition in the AES algorithm, Galois addition which is basically XOR operation is followed.

2. The Mix Columns stage in the AES encryption technique employs finite fields. The Advanced Encryption Standard (AES) employs the Galois Field GF (2^8), wherein every byte of the state matrix is regarded as an element within this finite field. The substitution and permutation operations of AES heavily rely on this particular element.

3.2 Blackbox AES 128 Implementation (ECB)

In the paper, FIPS 197 AES-128 is specified extensively [19] It is a 10-round algorithm that takes a 16-byte input and gives a 16-byte output using a 16-byte key. Sub Bytes, Shift Rows, Mix Columns, and Add Round Key are smaller, sub-algorithms that make up the Advanced Encryption Standard (AES) algorithm. Now a description of different sub-parts and the state of the algorithm is given below:

SubBytes: This function uses a substitution table named “Rijndael S Box”.The bytes of the state are replaced with the elements of the Sbox. If a code snippet is shown:

```
For i in range(0-9)
   State[i]=S[State[i]]
```

For the decryption process, there is the “inverse S box” which returns the same state that was substituted during the encryption process

ShiftRows: One of the simplest data scrambling operations is ”ShiftRows”. It entails shifting each row by a specific number of units, as the name suggests. Row n is specifically moved left by (n-1) units. Accordingly, the first row is left unchanged, the second row is left shifted one unit, the third row is shifted two units to the left, and finally the fourth row is shifted three units to the left.

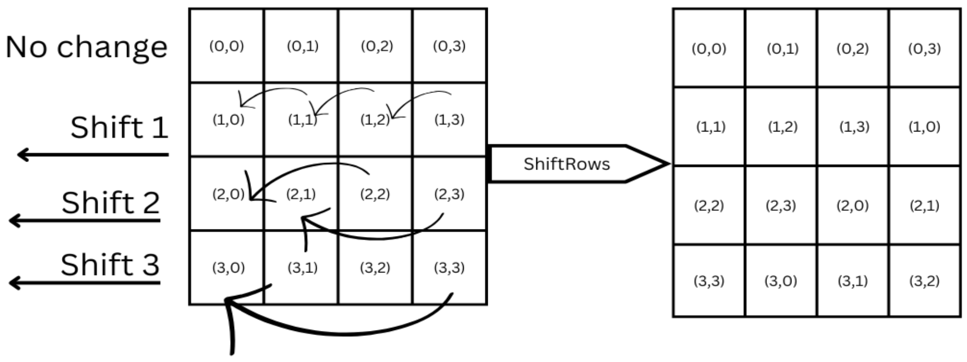


Figure 3.2.1: Shifting Rows process

MixColumns: This method concentrates on manipulating columns rather than rows. Every column in the state is transformed linearly by the ”MixColumns” method. Given that each column of the matrix or array has four rows, the matrix that has been used here also has a dimension of 4x4.

A matrix of dimension 4x4:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

The implementation of Blackbox in ECB mode is shown in Figure 3.2.2.

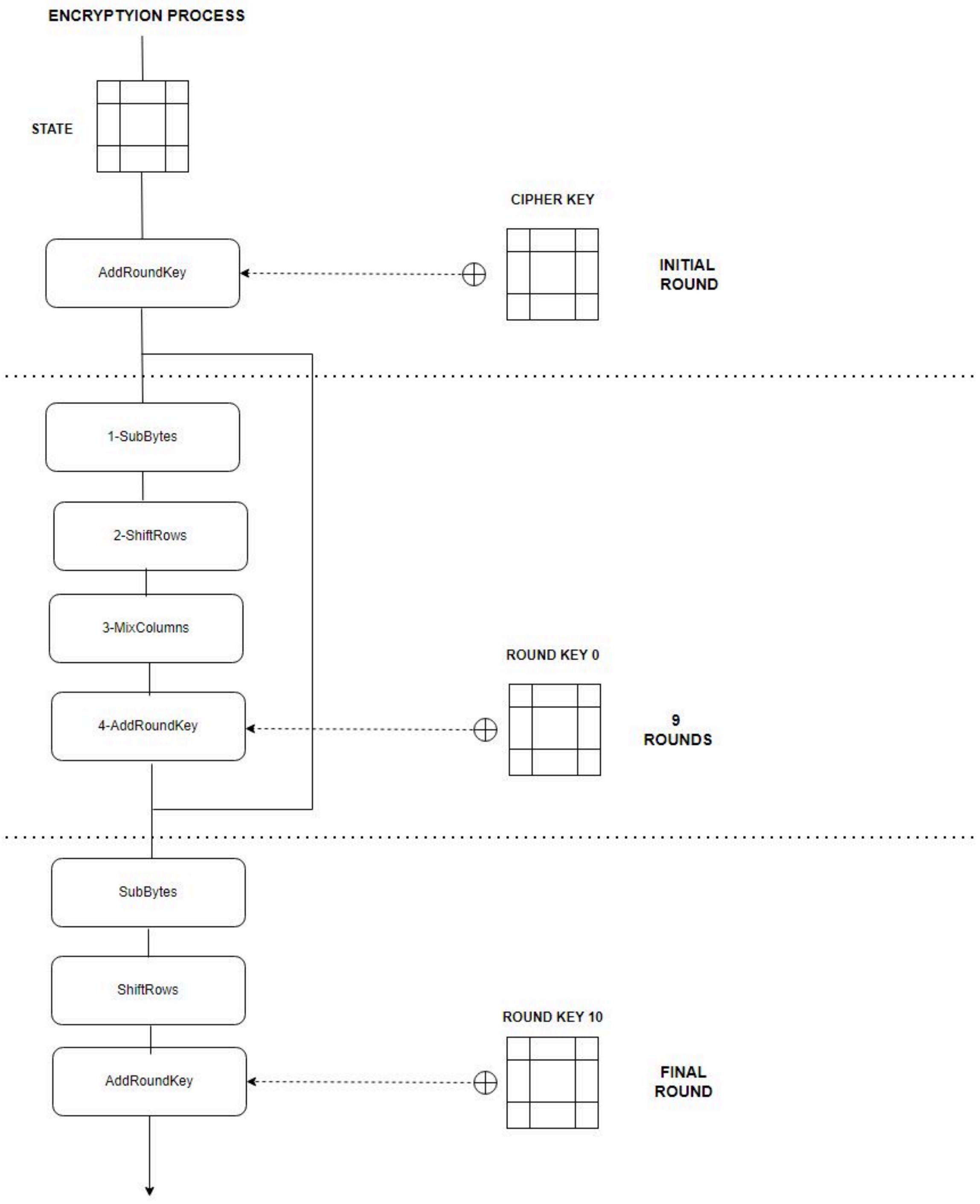


Figure 3.2.2: Implementation of Blackbox (ECB mode)

3.3 Modified Blackbox Implementations

During the above implementation, the AddRoundKey is done first in the initial phase and then the function is called 10 times in the next round. As a whole, a total of 11 times "AddRoundKey" is done in the algorithm. However, the loop can be reorganized to bring the AddRoundKey(state,k_0) into the 10 rounds and consequently, the 9th and 10th rounds of AddRoundKey(state,k_9), AddRoundKey(state,k_10)

will be done in the final round.
Moreover, SubBytes preceded by shift rows give the same result as the result of shift rows preceded by subtypes
So the whole process looks like :

Algorithm 1 AES Encryption Algorithm

```
 1: for i from 0 to 9 do
 2:     AddRoundKey(state, key_i)
 3:     ShiftRows(state)
 4:     SubBytes(state)
 5:     MixColumn(state)
 6: end for
 7:  10th round
 8: AddRoundKey(state, key_9)
 9: ShiftRows(state)
10: SubBytes(state)
11: AddRoundKey(state, key_10)
```

3.4 Extended Blackbox Implementation

Rearranging operations, the code snippet should look like this:

Algorithm 2 Rearranging Operations of AES Encryption Algorithm

```
 1: for i in range(10) do
 2:     ShiftRows(state)
 3:     AddRoundKey(state, shifted_key_i)
 4:     SubBytes(state)
 5:     MixColumn(state)
 6: end for
 7:  10th round
 8: ShiftRows(state)
 9: AddRoundKey(state, shifted_key_9)
10: SubBytes(state)
11: AddRoundKey(state, key_10)
```

AddRoundKey preceded by ShiftRows gives the same result as the result of ShiftRows preceded by AddRoundKey since this is a linear transformation. In this case, the shifting of the key is not done automatically rather it has to be done in the AddRoundKey step.
However, in this modification of Blackbox, only the last AddRounKey(state,key_10) does not need any shifting of keys.

3.5 Counter Mode Blackbox AES (CTR)

This mode of encryption is very parallelizable because each block of plaintext is separately encrypted. As a consequence, the implementation of hardware and software in an effective way is possible through this mode. Moreover, when compared to other modes like CBC, CTR mode is more effective in terms of data expansion because it does not require padding of the plaintext. Not only that, random access to data is possible only through this mode. As a consequence, a specific portion of the data can be decrypted through this process. The steps of CTR mode are:

Key and IV selection: A block cipher's secret encryption key has to be selected such as AES-128, or AES-256. Afterward, a nonce or an initialization vector (IV). has to be chosen which should either be a nonce that is never repeated with the same key or a random number for each encryption session.

Counter selection: For the next part a counter has to be selected and the counter's initial value has to be set which is typically 0. After that,1 is added to the counter for each block of plaintext that is presented and the resulting value is converted into binary.

Encryption: The selected key and block cipher have to be used to encrypt the counter value. The result is a block of pseudorandom bits known as the" keystream."

XOR of keystream and plaintext: XOR operation between the keystream which was got in the previous step and the block of pain text has to be done and this xor will give the ciphertext for the respective block.

Repeatation of steps: The step "Encryption" and "XOR" has to be done repeatedly until the full message is encrypted.

Concatenation of the ciphertexts: At last, all the ciphertexts have to be concatenated to make the final encrypted message.

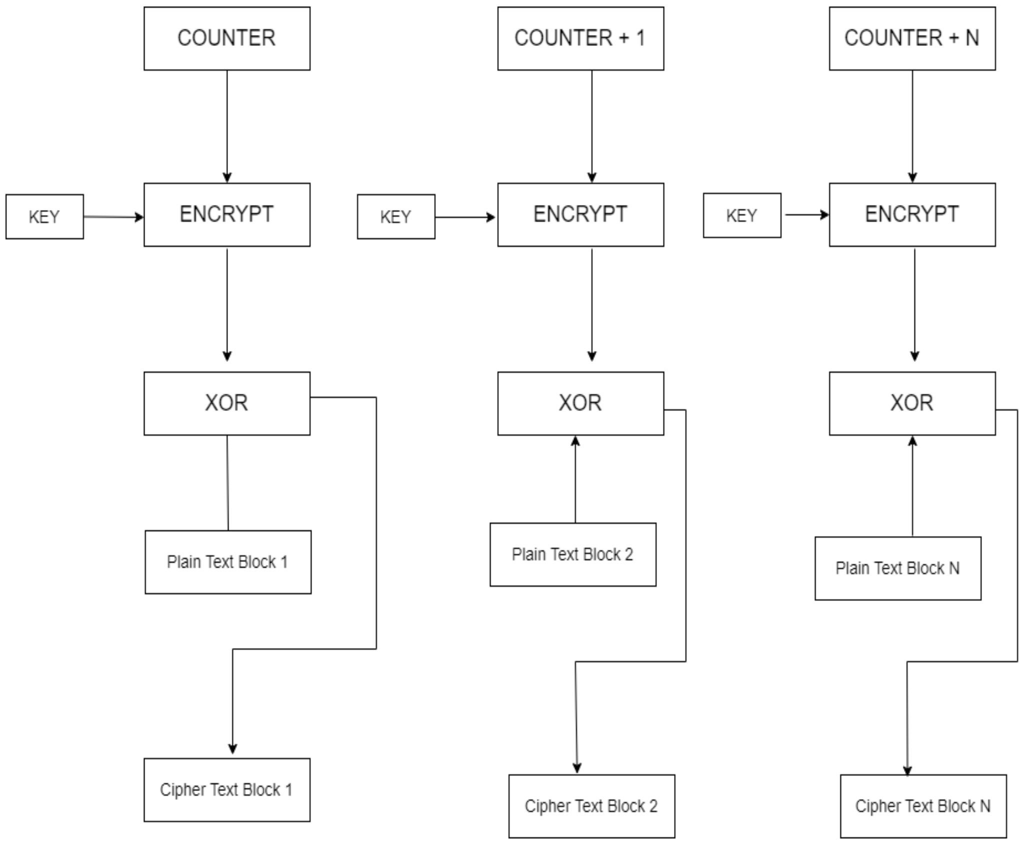


Figure 3.5.3: Blackbox with CTR operation

The Blackbox algorithm irrespective of CTR mode or ECB mode is only concerned about the input and output, and in this system key is totally exposed. If an attacker knows about the key, he can easily apply reverse engineering to find out the actual plaintext. So, the concern is to protect the key and make the algorithm more secure. As a consequence, a new approach of encryption came where the key was embedded in the algorithm and this approach gave inspiration to the implementation of an algorithm that will transform Blackbox cryptographic algorithm to the Whitebox cryptographic algorithm.

Chapter 4

Implementation of Whitebox AES

4.1 Table-Based Whitebox Implementation

At the outset, encryption adheres to the blackbox model where the attacker can only observe the input and the output. However, it has been observed that the key is compromised in this model. The attacker can guess the key with a time complexity of 2^{30} [15]. To address this vulnerability, the Blackbox model is transferred into a white box model so that the encryption model can fit even if the attacker can see the initial steps. This consideration is because if the attacker can monitor the initial steps it becomes easier to guess the key. Therefore, the key must be kept hidden to make computation harder for the attacker.

4.1.1 T-box Generation Process

Whitebox cryptography employs various techniques to protect the encryption key, making it resistant to various attacks, even when an attacker has complete knowledge of the encryption algorithm and operations. The main goal is to conceal the direct operations through the use of various lookup tables while keeping the main function as it is.
In the final modification of the blackbox model,

Algorithm 3 Final Modification of Blackbox Model

```
1: for $i$ in range(0-9) do
2:     ShiftRows(state)
3:     AddRoundKey(state, shifted_key_i)
4:     SubBytes(state)
5:     MixColumn(state)
6: end for
7: 10th round:
8: ShiftRows(state)
9: AddRoundKey(state, shifted_key_9)
10: SubBytes(state)
11: AddRoundKey(state, key_10)
```

For every round AddRoundKey and SubBytes can be replaced with the help of 16 look-up tables which are called T-boxes. Moreover, the lookup table consists of every possible combination of AddRoundKey and SubBytes. There needs to be two separate lookup tables, one for rounds 1 to 9 and the other for the last round.

4.1.2 T-box Generation for Rounds 1-9

$$T_i^r(x) = S(x \oplus b_k^{(r-1)}[i])$$

A 3D array is declared, where the size of the array is 'Tboxtable [9] [16] [256]'. There need to be 3 nested loops. The first loop runs for 9 iterations. For every 9 rounds, the key is fixed after being generated from the KeyExpansion method. Moreover, the second loop runs for 16 rounds, representing the 16 hexadecimal bytes of the key. As 1 hex byte represents 8 bits of binary numbers, there are a total of 256 combinations iterating at the third loop.
In the second loop, it iterates and takes one hex byte, passing it as a parameter to the third loop, which calculates every possible combination to add it to the hex key and substitutes the addition value using a substitution array. Thus a lookup table works for the AddRoundKey and Subbytes.

Note that in every iteration the key of 16 hex byte is shifted one bit.

4.1.3 T-box Generation for Final Round

In the final round, addround key operation is executed two times making the final round an exception from other rounds.

$$T_{10}^{i}(x) = S(x \oplus b_k^9[i]) \oplus k_{10}[i]$$

It works the same as before, running three nested loops with the additional step of XOR-ing the unshifted last sequence of keys.

4.1.4 Ty Table Generation

In the mentioned sequence, MixColumns is followed by AddRoundKey and SubBytes. Instead of direct operations, MixColumns computations are executed using tables, which confuses the attackers when trying to guess the similarity between the inputs and outputs.

For each round, the present state of 16 hexadecimal bytes is divided into T_0, T_1, T_2, and T_3, interpreted as a column vector. Lets the present state be,

$$\begin{bmatrix} \mathrm{S}_0 & \mathrm{S}_1 & \mathrm{S}_2 & \mathrm{S}_3 \\ \mathrm{S}_4 & \mathrm{S}_5 & \mathrm{S}_6 & \mathrm{S}_7 \\ \mathrm{S}_8 & \mathrm{S}_9 & \mathrm{S}_{10} & \mathrm{S}_{11} \\ \mathrm{S}_{12} & \mathrm{S}_{13} & \mathrm{S}_{14} & \mathrm{S}_{15} \end{bmatrix}$$

S_0	S_4	S_8	S_{12}	S_1	S_5	S_9	S_{13}	S_2	S_6	S_{10}	S_{14}	S_3	S_7	S_{11}	S_{15}
T_0				T_1				T_2				T_3			

After that, the T Tables are formed by multiplying the T_0, T_1, T_2, and T_3 with the respective columns of the MC matrix.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

$$\begin{bmatrix} T_0 \\ T_1 \\ T_2 \\ T_3 \end{bmatrix} = \mathrm{T}_0 \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus \mathrm{T}_1 \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus \mathrm{T}_2 \begin{bmatrix} 01 \\ 02 \\ 02 \\ 01 \end{bmatrix} \oplus \mathrm{T}_3 \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix}.$$

This equation can be broken down into four terms of XOR relation between them. Furthermore, T_0, T_1, T_2, T_3 are the parameters to search in the lookup table for their 256 combinations. The modified equations are:

$$\begin{aligned} \mathrm{T}y_0(\mathrm{s}) &= \mathrm{T}_0.\begin{bmatrix} 02 & 03 & 01 & 01 \end{bmatrix}^T \\ \mathrm{T}y_1(\mathrm{s}) &= \mathrm{T}_1.\begin{bmatrix} 03 & 02 & 01 & 01 \end{bmatrix}^T \\ \mathrm{T}y_2(\mathrm{s}) &= \mathrm{T}_2.\begin{bmatrix} 01 & 03 & 02 & 01 \end{bmatrix}^T \\ \mathrm{T}y_3(\mathrm{s}) &= \mathrm{T}_3.\begin{bmatrix} 01 & 01 & 03 & 02 \end{bmatrix}^T \end{aligned}$$

Finally, this all four equations ended up to creating 144 tables coming 4 copies from 9 rounds. The final equation is as follows:

$$\mathrm{T}y_0(T_0) \oplus \mathrm{T}y_1(T_1) \oplus \mathrm{T}y_2(T_2) \oplus \mathrm{T}y_3(T_3)$$

4.1.5 XOR Table Construction

The XOR lookup table serves the purpose to find every possible xor combination of two 8-bit binary numbers. It takes two 8 bit binary numbers as input and gives a 8 bit binary number of their xor results. Furthermore, when implementing this xor lookup table in code there will be simply two nested loops running 256 times. (256 possible combinations for a 8-bit binary number.

$$\text{XOR Table}[i][j] = (\text{BinaryNumber1} \oplus \text{BinaryNumber2});$$

4.1.6 Function Conversion Process

The traditional sequence of AES in Blackbox model -

Algorithm 4 AES Encryption Algorithm

```
1: for i in range(0-9) do
2:     AddRoundKey(state, key_i)
3:     ShiftRows
4:     SubBytes
5:     MixColumn
6: end for
7: 10th round:
8: AddRoundKey(state, shifted_key_9)
9: ShiftRows
10: SubBytes
11: AddRoundKey(state, key_10)
```

To combine the operations and integrate them into lookup tables the sequences needed to be changed so that the output remains unchanged. This can be achieved by reordering the linear operations since the permutation of these operations does not alter the output. The final sequence for transforming it into a whitebox model is as follows:

Algorithm 5 Modified AES Encryption Algorithm for Whitebox Model

```
1: for i in range(0-9) do
2:     ShiftRows
3:     AddRoundKey(state, shifted_key_i)
4:     SubBytes
5:     MixColumn
6: end for
7: 10th round:
8: ShiftRows
9: AddRoundKey(state, shifted_key_9)
10: SubBytes
11: AddRoundKey(state, key_10)
```

Working with this sequence, the ShiftRows operation remains in its position. On the other hand, the AddRoundKey and SubBytes operations combine to obtain output from Tbox lookup tables. Furthermore, the MixColumns operation is replaced by Ty lookup tables. Lastly, these two combine to create a function called T-Box-Ty Tables. XOR is followed by TBox Ty Tables.

Algorithm 6 Whitebox Transformation Algorithm

```
1: for i in range(0-9) do
2:     ShiftRows
3:     TBoxesTyTables
4:     XORTables
5: end for
6: 10th round:
7: ShiftRows
8: TBoxes
9: Ciphertext
```

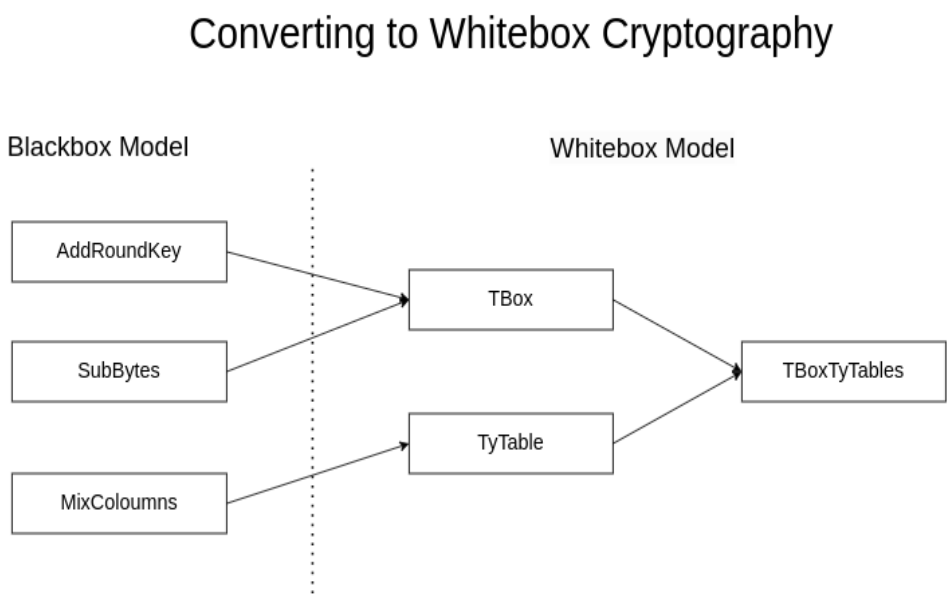


Figure 4.1.1: Converting Blackbox to Whitebox Implementation

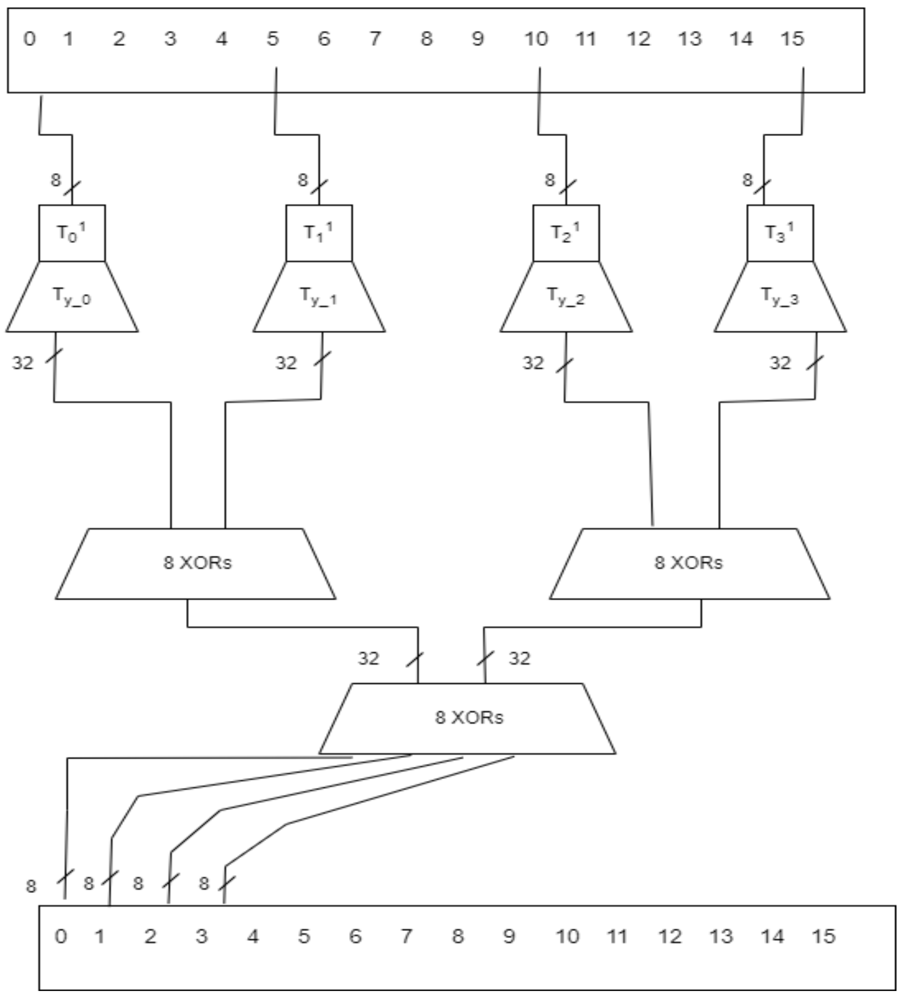


Figure 4.1.2: Data flow of Whitebox implementation

From the implementation illustrated in Figure 4.1.2, the exposure of the key was reduced and the code was obfuscated to a certain extent. Now if an attacker wants to extract the key, he has to go through a long range of permutations and combinations from the lookup tables. Apart from the lookups, another security dilemma arrived where the knowledge about the entries of the composed Tbox Ty tables made way to certain side-channel attacks and reverse engineering processes which gave birth to extra security layer implementation known as implementation with encoding

4.2 Secured Whitebox Implementation

If a TBox or Ty Table entry is known to an attacker, they can easily apply reverse engineering to acquire the key. Something must be done to protect the composed TBox Ty Tables so that applying reverse engineering or any other tracing does not extract the key. That is why, Chow et al proposed a technique that involves applying encodings to the implementation. [10] The encoding is applied in the two following ways -

1. Internal Encoding

2. External Encoding

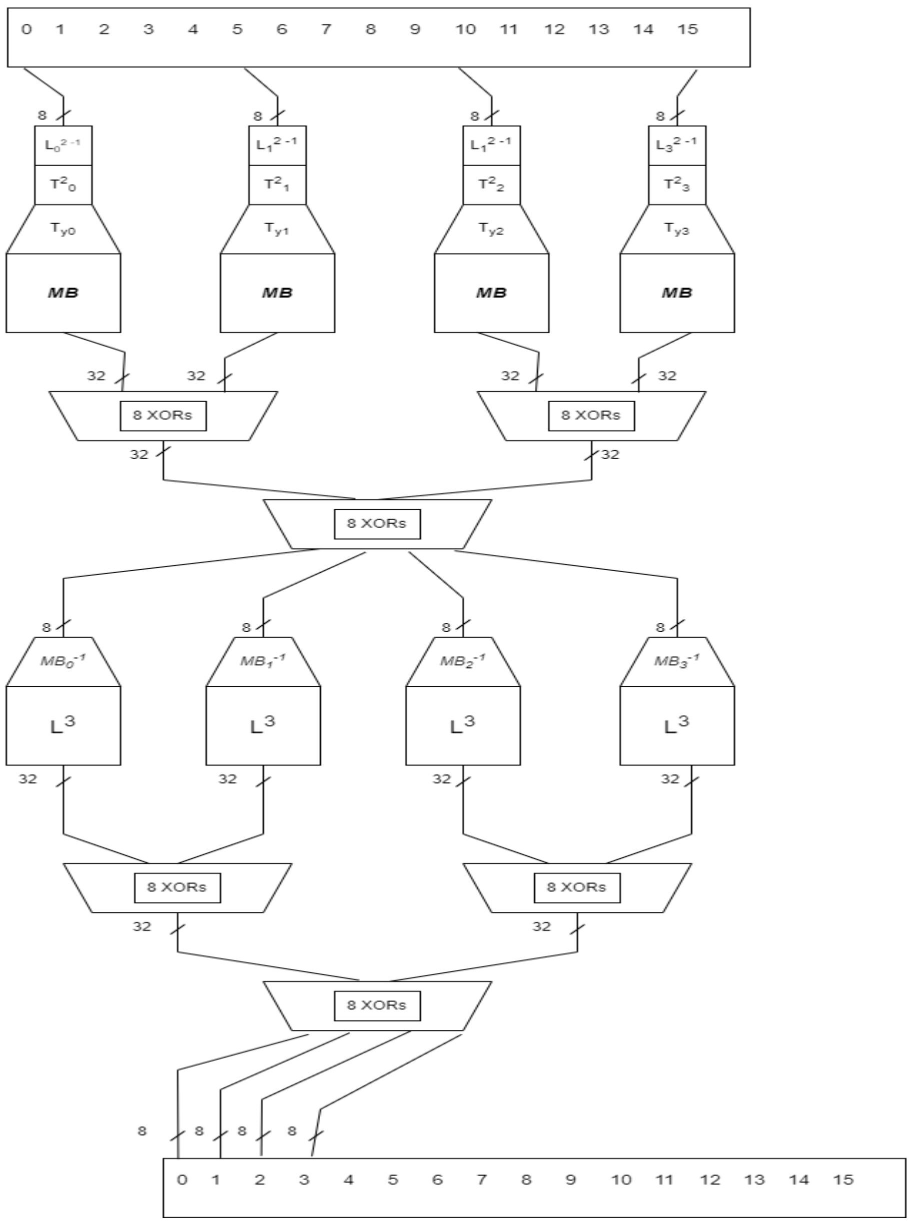


Figure 4.2.3: Workflow of Whitebox AES implementation with added layer of encoding in Chapter

4.2.1 Internal Encoding Mechanism

The whole process of internal encoding is applied to the composed TBox Ty Tables. The bytes of round keys are incorporated into the composed TBox Ty Tables. These lookup tables can be considered miniature block ciphers. An internal encoding is applying bijection to these block ciphers.

The bijection is simply a function that has one-to-one and onto property. It can be said that bijections are invertible, they are cardinal and preserve order. Let, MB is a bijection function that will be applied to the composed TBox Ty Tables. The whole process of internal encoding is done in the following two ways-

1. From rounds 1 to 9 this MB will be applied to the output of the tables. As a result, in every round, the output will be different than the previous table-based implementation to ensure obfuscation.

2. Now, let L_0^2, L_1^2, L_2^2, L_3^2 be the 8 bit to 8-bit bijection functions applied in round 2. From round 2 to round 10, the inverse of these four functions will be applied to the input of the tables so that in every round the actual state that has to be XORed with the key goes into action for lookup.

The action can be represented as the following in round 1:

Before applying encoding:

$$\begin{gathered}\mathrm{T}_{y0} \circ T_0^1 \\ \mathrm{T}_{y_1} \circ T_1^1 \\ \mathrm{T}_{y_2} \circ T_2^1 \\ \mathrm{T}_{y_3} \circ T_3^1\end{gathered}$$

After applying encoding:

$$\begin{gathered}MB \circ T_{y_0} \circ T_0^1 \\ MB \circ T_{y_1} \circ T_1^1 \\ MB \circ T_{y_2} \circ T_2^1 \\ MB \circ T_{y_3} \circ T_3^2\end{gathered}$$

In Round 2 the action can be presented in the following way:

Before applying encoding:

$$\begin{gathered}\mathrm{T}_{y0} \circ T_0^2 \\ \mathrm{T}_{y_1} \circ T_1^2 \\ \mathrm{T}_{y_2} \circ T_2^2 \\ \mathrm{T}_{y_3} \circ T_3^2\end{gathered}$$

After applying encoding:

$$\begin{gathered}MB \circ T_{y_0} \circ T_0^2 \circ L_0^{2-1} \\ MB \circ T_{y_1} \circ T_1^2 \circ L_1^{2-1} \\ MB \circ T_{y_2} \circ T_2^2 \circ L_2^{2-1} \\ MB \circ T_{y_3} \circ T_3^2 \circ L_3^{2-1}\end{gathered}$$

Also in round 10, there is only input bijection.

The flow of round 9 and 10 is -

$$MB \circ T_{y0} \circ T_0^9 \circ L_0^{9-1}$$
$$MB \circ T_{y1} \circ T_1^9 \circ L_1^{9-1}$$
$$MB \circ T_{y2} \circ T_2^9 \circ L_2^{9-1}$$
$$MB \circ T_{y3} \circ T_3^9 \circ L_3^{9-1}$$

Round 10 after applying encoding:

$$\mathrm{T}_{y0} \circ T_0^{10} \circ L_0^{10-1}$$
$$\mathrm{T}_{y1} \circ T_1^{10} \circ L_1^{10-1}$$
$$\mathrm{T}_{y2} \circ T_2^{10} \circ L_2^{10-1}$$
$$\mathrm{T}_{y3} \circ T_3^{10} \circ L_3^{10-1}$$

4.2.2 External Encoding Mechanism

The concept of extracting the cipher key to break the security of block cipher became less vulnerable when there came software that could decrypt the ciphertext. As a consequence, there came an answer from Chow et al made implementations that would map encoded ciphertext to encoded plaintext instead of raw ciphertext and plaintext. As a whole, the implementations that affect the input and output encoding of the ciphertext are referred to as external encodings.

Here, the concept and the implementation are the same as internal encoding. The only difference is that in internal encodings the encoding scheme is applied to the block ciphers on the other hand, in external encoding the scheme is applied to the ciphertexts. The process of external encoding in round1 is given below:

Here

RT = Raw text and
CT = Cipher text.

Before applying encoding:

$$RT — CT_1$$

After applying encoding:

$$RT — CT_1$$
$$CT_1 — CT_1\ (Encoded)$$

In round 2 the action can be presented in a similar way:

Before applying encoding:

$$CT_1 — CT_2$$

After applying encoding:

$$CT_1(\mathit{Encoded}) — CT_1$$
$$CT_1 — CT_2$$
$$CT_2 — CT_2(\mathit{Encoded})$$

Finally, during round 10 the process is similar to internal encoding.
Before applying encoding:

$$CT_9 — CT_{10}$$

After applying encoding:

$$CT_9(\mathit{Encoded}) — CT_9$$
$$CT_9 — CT_{10}$$

When the composed lookup tables are protected with encoding, there are too many constructions for an attacker. By the list of table constructions, an attacker may apply reverse engineering to study the list of table constructions. As the entries of the tables are manipulated, it will not be possible to apply these techniques to deduce the key byte.

4.3 Counter Mode Whitebox AES (CTR)

The main adverse effect of whitebox implementation is that generally, it requires a lot of table lookups, and also after the secured implementation of internal and external encoding, the runtime complexity of code increases to a significant extent. As a whole, software obfuscation techniques aimed at protecting the security of the implementation make the encryption process much slower. As a consequence, the concept of counter mode implies that the whitebox algorithm is used to encrypt only part of the message and the rest of the message is encrypted using any classical algorithm. Moreover, almost all data protecting mechanisms such as SSL, TLS, and SSH are based on a shared secret which is known as a session key. Designers who are after the solution management of software want to apply this session key in whitebox encryption. However, the direct usage of this key is not possible in the whitebox scheme as whitebox is slow and is separated from an environment that is running. Hence, the main objective is to apply the session key directly in the components of the scheme except for the implementation of the whitebox algorithm. Therefore, in the counter mode, the nonce will be encrypted by a whitebox algorithm, and use the nonce as a replacement for the one-time key.

The implementation is done by using two separate keys - one for the whitebox primitive and the other for the encryption algorithm. Here the whitebox encryption algorithm is used for the encryption of nonce and the encryption algorithm is used for the encryption of the plaintext. Again the keys k_1 and k_2 used in this algorithm are fixed but the nonce is changeable in every encryption session. [18]

Again, the scheme limits limited encryption of messages of length at most 2^{64} blocks in a single session. For any practical purpose, this amount is sufficient. Moreover, unlike other nonce based algorithms, the reuse of nonce is not allowed. As far as security is concerned, data complexity of 2^{64} and time complexity of 2^{80}.As a result, if anyone wants to extract the key or recover part of the plain text, would require either 2^{64} messages or 2^{80} memory. [18]

Workflow of counter mode whitebox implementation:

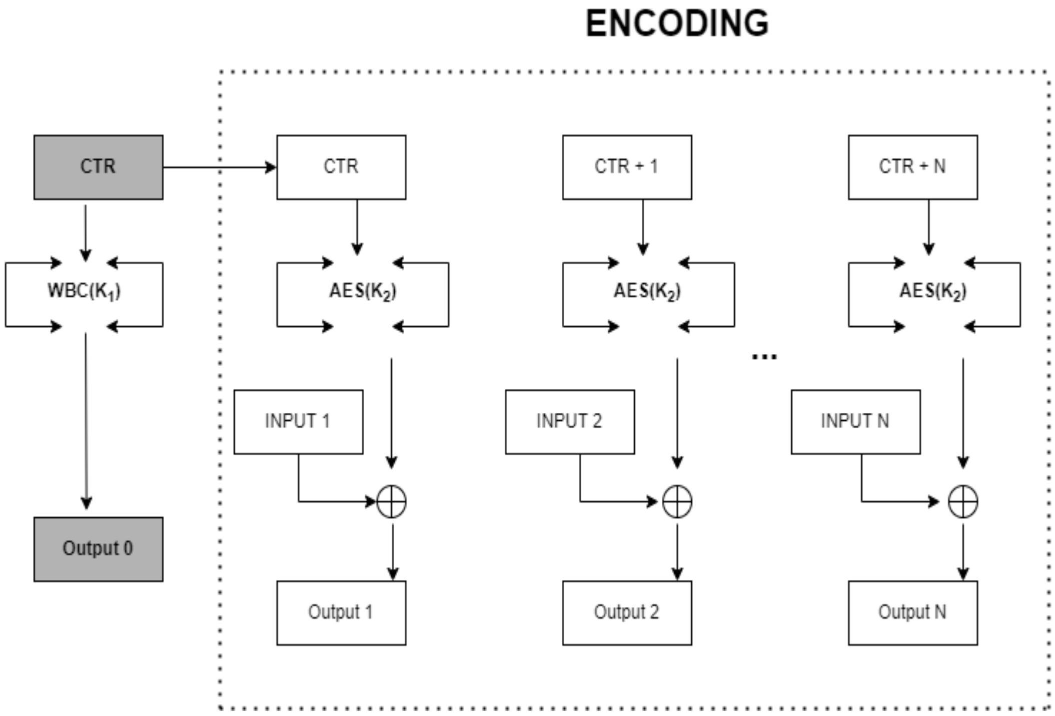


Figure 4.3.4: CTR-WBC: A hybrid Whitebox scheme with CTR operation in Chapter

4.4 Summary

Once all the lookup tables were implemented, the meomory usage can be figured out. This involves looking at how well the tables fit in and how they affect the overall memory.

Array	Dimensions	Size of Each Element (bytes)	Total Size (bytes)
xorTable	256×256	1	$256 \times 256 \times 1$
Tbox_round	$9 \times 16 \times 256$	1	$9 \times 16 \times 256 \times 1$
Ty_Table	$4 \times 256 \times 4$	1	$4 \times 256 \times 4 \times 1$
Tbox_final	$1 \times 16 \times 256$	1	$1 \times 16 \times 256 \times 1$
Total Memory Size	-	-	Sum of the above sizes

$$\text{Total Memory Size} = 65536 + 36864 + 4096 + 4096 = 110592 \text{ bytes}$$

$$\text{Total Memory Size (MB)} = \frac{110592 \text{ bytes}}{1024^2} \approx 0.105 \text{ MB}$$

Chapter 5

Security and Performance Analysis

5.1 Performance Analysis of Blackbox and Whitebox Implementations

During the implementation, a runtime analysis was done of the ECB mode, CTR mode, and also the runtime of Whitebox implementation. CTR mode allows for parallel encryption and decryption of blocks, as each block is independently XOR with the output of the counter mode encryption. On the other hand, ECB mode of encryption uses a uniform key to individually encrypt each block. As a consequence, in some cases, CTR mode is faster than ECB mode. In addition, the CTR mode is highly suitable for facilitating random access to encryption. But in the case of ECB mode random access of data is not possible. Moreover, ECB mode adds padding to data if the data is not an exact multiple of the block size. In the case of CTR mode padding is not done.

Now as far as the scenario of Whitebox Implementation is concerned, the conversion algorithm needs, 160 Tboxes, 144 Ty tables, and 864 XOR tables which increases the space complexity and also the memory complexity. As a whole, the computation of the lookup tables increases the runtime to an extent. That is why, the runtime of Whitebox implementation is a bit higher than the Blackbox implementation.

5.2 Security Analysis

Security analysis will have a series of observations.

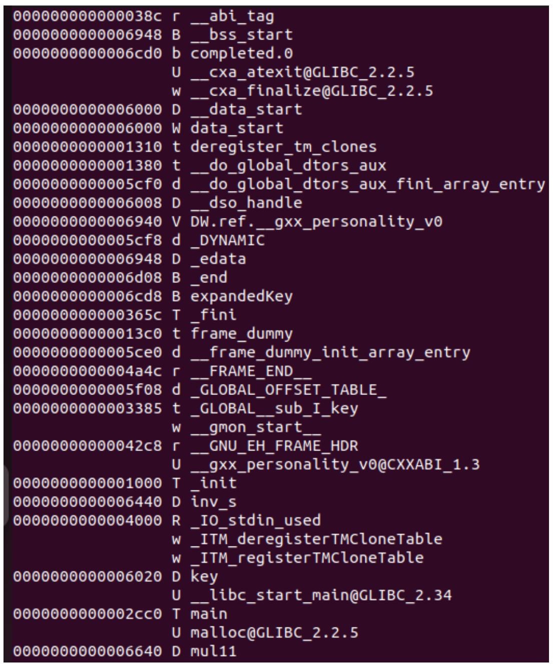
The data flow of the security analysis -

$$\begin{tikzcd}
\text{Symbol Table Generation} \arrow[d] \\
\text{Behaviour Integration} \arrow[d] \\
\text{Symbol Table Stripping} \arrow[d] \\
\text{Again Behaviour Integration of the graph} \arrow[d] \\
\text{Behaviour Randomization of the graph}
\end{tikzcd}$$

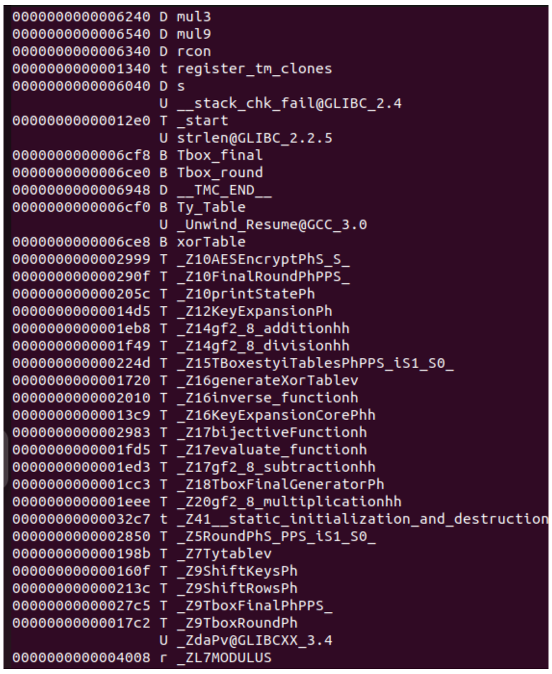
Symbol Table Generation: A symbol table is a data structure to store information about variables, functions, and other essentials of a code. It aids in managing and organizing the identifiers and variables and thus is a crucial part of a program. As a whole, the purpose of the symbol table is to ensure the variable and function management and the scope management. The information stored in the symbol table is identifier name, data type, memory location, scope information, additional attributes, etc. Moreover, the symbol table also helps in lifetime management by integrating the concept of local variables and global variables, symbol resolution during the compilation process, and error detection by detecting undeclared variables.

For the generation of a symbol table, a command-line utility in Linux is used which is known as the 'nm' tool. Here, the 'nm' refers to the 'name list.'This standard Unix command is used mainly for examining the behaviors of the symbol table of binary executable files. Moreover, this tool can help in symbol exploration, debugging information, and library analysis.

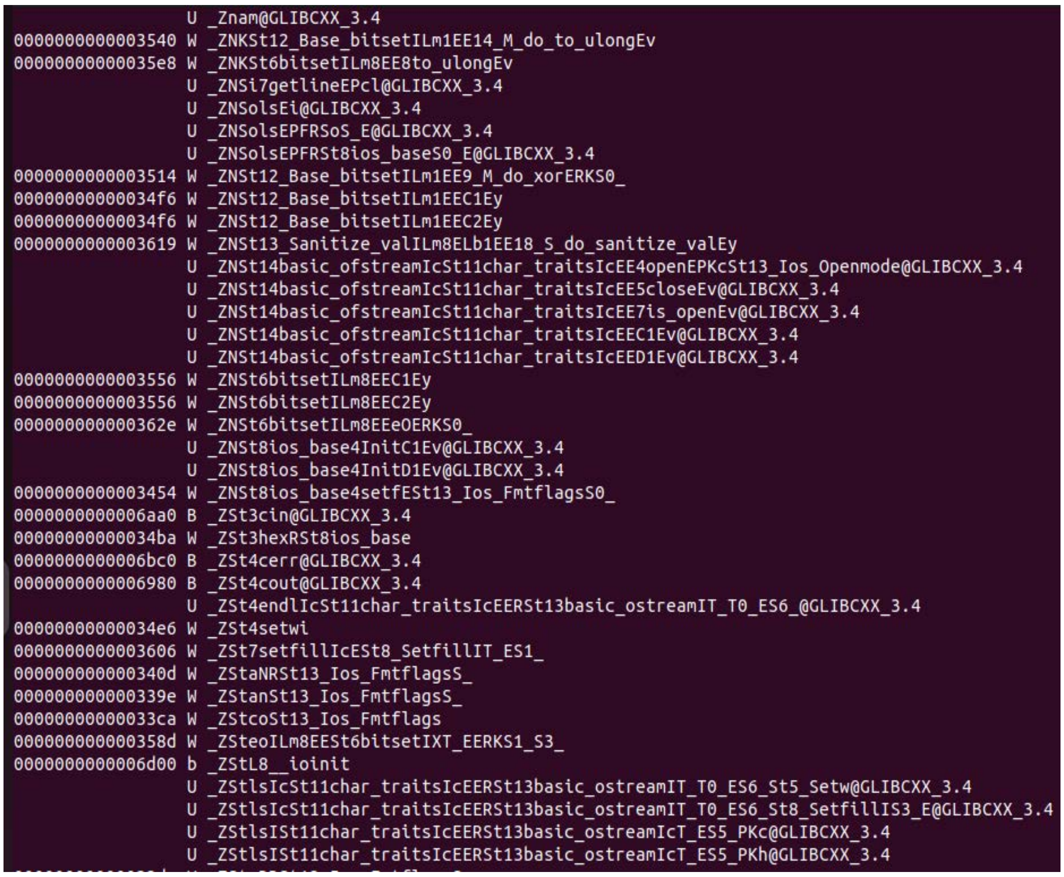
A visualization of the symbol table which was generated for the implementation of AES-128 whitebox implementation is shown in Figure 5.2.2.



```
000000000000038c r __abi_tag
0000000000006948 B __bss_start
0000000000006cd0 b completed.0
                 U __cxa_atexit@GLIBC_2.2.5
                 w __cxa_finalize@GLIBC_2.2.5
0000000000006000 D __data_start
0000000000006000 W data_start
0000000000001310 t deregister_tm_clones
0000000000001380 t __do_global_dtors_aux
0000000000005cf0 d __do_global_dtors_aux_fini_array_entry
0000000000006008 D __dso_handle
0000000000006940 V DW.ref.__gxx_personality_v0
0000000000005cf8 d _DYNAMIC
0000000000006948 D _edata
0000000000006d08 B _end
0000000000006cd8 B expandedKey
000000000000365c T _fini
00000000000013c0 t frame_dummy
0000000000005ce0 d __frame_dummy_init_array_entry
0000000000004a4c r __FRAME_END__
0000000000005f08 d _GLOBAL_OFFSET_TABLE_
0000000000003385 t _GLOBAL__sub_I_key
                 w __gmon_start__
00000000000042c8 r __GNU_EH_FRAME_HDR
                 U __gxx_personality_v0@CXXABI_1.3
0000000000001000 T _init
0000000000006440 D inv_s
0000000000004000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000006020 D key
                 U __libc_start_main@GLIBC_2.34
0000000000002cc0 T main
                 U malloc@GLIBC_2.2.5
0000000000006640 D mul11
```



```
0000000000006240 D mul3
0000000000006540 D mul9
0000000000006340 D rcon
0000000000001340 t register_tm_clones
0000000000006040 D s
                 U __stack_chk_fail@GLIBC_2.4
00000000000012e0 T _start
                 U strlen@GLIBC_2.2.5
0000000000006cf8 B Tbox_final
0000000000006ce0 B Tbox_round
0000000000006948 D __TMC_END__
0000000000006cf0 B Ty_Table
                 U _Unwind_Resume@GCC_3.0
0000000000006ce8 B xorTable
0000000000002999 T _Z10AESEncryptPhS_S_
000000000000290f T _Z10FinalRoundPhPPS_
000000000000205c T _Z10printStatePh
00000000000014d5 T _Z12KeyExpansionPh
0000000000001eb8 T _Z14gf2_8_additionhh
0000000000001f49 T _Z14gf2_8_divisionhh
000000000000224d T _Z15TBoxestyiTablesPhPPS_iS1_S0_
0000000000001720 T _Z16generateXorTablev
0000000000002010 T _Z16inverse_functionh
00000000000013c9 T _Z16KeyExpansionCorePhh
0000000000002983 T _Z17bijectiveFunctionh
0000000000001fd5 T _Z17evaluate_functionh
0000000000001ed3 T _Z17gf2_8_subtractionhh
0000000000001cc3 T _Z18TboxFinalGeneratorPh
0000000000001eee T _Z20gf2_8_multiplicationhh
00000000000032c7 t _Z41__static_initialization_and_destruction
0000000000002850 T _Z5RoundPhS_PPS_iS1_S0_
000000000000198b T _Z7Tytablev
000000000000160f T _Z9ShiftKeysPh
000000000000213c T _Z9ShiftRowsPh
00000000000027c5 T _Z9TboxFinalPhPPS_
00000000000017c2 T _Z9TboxRoundPh
                 U _ZdaPv@GLIBCXX_3.4
0000000000004008 r _ZL7MODULUS
```



```
                 U _Znam@GLIBCXX_3.4
0000000000003540 W _ZNKSt12_Base_bitsetILm1EE14_M_do_to_ulongEv
00000000000035e8 W _ZNKSt6bitsetILm8EE8to_ulongEv
                 U _ZNSi7getlineEPcl@GLIBCXX_3.4
                 U _ZNSolsEi@GLIBCXX_3.4
                 U _ZNSolsEPFRSoS_E@GLIBCXX_3.4
                 U _ZNSolsEPFRSt8ios_baseS0_E@GLIBCXX_3.4
0000000000003514 W _ZNSt12_Base_bitsetILm1EE9_M_do_xorERKS0_
00000000000034f6 W _ZNSt12_Base_bitsetILm1EEC1Ey
00000000000034f6 W _ZNSt12_Base_bitsetILm1EEC2Ey
0000000000003619 W _ZNSt13_Sanitize_valILm8ELb1EE18_S_do_sanitize_valEy
                 U _ZNSt14basic_ofstreamIcSt11char_traitsIcEE4openEPKcSt13_Ios_Openmode@GLIBCXX_3.4
                 U _ZNSt14basic_ofstreamIcSt11char_traitsIcEE5closeEv@GLIBCXX_3.4
                 U _ZNSt14basic_ofstreamIcSt11char_traitsIcEE7is_openEv@GLIBCXX_3.4
                 U _ZNSt14basic_ofstreamIcSt11char_traitsIcEEC1Ev@GLIBCXX_3.4
                 U _ZNSt14basic_ofstreamIcSt11char_traitsIcEED1Ev@GLIBCXX_3.4
0000000000003556 W _ZNSt6bitsetILm8EEC1Ey
0000000000003556 W _ZNSt6bitsetILm8EEC2Ey
000000000000362e W _ZNSt6bitsetILm8EEeOERKS0_
                 U _ZNSt8ios_base4InitC1Ev@GLIBCXX_3.4
                 U _ZNSt8ios_base4InitD1Ev@GLIBCXX_3.4
0000000000003454 W _ZNSt8ios_base4setfESt13_Ios_FmtflagsS0_
0000000000006aa0 B _ZSt3cin@GLIBCXX_3.4
00000000000034ba W _ZSt3hexRSt8ios_base
0000000000006bc0 B _ZSt4cerr@GLIBCXX_3.4
0000000000006980 B _ZSt4cout@GLIBCXX_3.4
                 U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@GLIBCXX_3.4
00000000000034e6 W _ZSt4setwi
0000000000003606 W _ZSt7setfillIcESt8_SetfillIT_ES1_
000000000000340d W _ZStaNRSt13_Ios_FmtflagsS_
000000000000339e W _ZStanSt13_Ios_FmtflagsS_
00000000000033ca W _ZStcoSt13_Ios_Fmtflags
000000000000358d W _ZSteoILm8EESt6bitsetIXT_EERKS1_S3_
0000000000006d00 b _ZStL8__ioinit
                 U _ZStlsIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_St5_Setw@GLIBCXX_3.4
                 U _ZStlsIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_St8_SetfillIS3_E@GLIBCXX_3.4
                 U _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@GLIBCXX_3.4
                 U _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKh@GLIBCXX_3.4
```

Figure 5.2.1: Visualization of the symbol table in Chapter

Where

U = Undefined symbols.
T or t = Text symbols.
D or d = Data symbols.
B or b = BSS section.
W or w = Weak symbols.
A or a = Absolute symbols.
C or c = Common symbols.

Behavior integration of the programme using Tracegraph: The binary executable of an implementation of a cryptographic algorithm helps in assess the security of the implementation. By executing the binary on a CPU of the corresponding architecture and observing its power consumption, a differential power analysis (DPA) can be done. But in this analysis, there is a matter of noise as the implementation is observed with a hardware architecture. However, in the whitebox attack model, one can do the analysis without any measurement noise. For such a level of observation, one can instrument the binary or instrument an emulator being in charge of the binary. While integrating the behavior of our program the first option was followed in the paper and while doing so some of the available dynamic binary instrumentation (DBI) frameworks were used. If the mechanism of DBI is considered, it uses the binary executable to analyze the bytecode of a virtual machine using a technique known as just-in-time compilation. This recompilation allows a transformation to be applied to the code while keeping the original computational effect the same. Some of the DBI frameworks such as pin and Valgrind allow adding callbacks in between the instructions of machine code by writing plugins or tools that integrate the process of recompilation. As a result, these specific callbacks help generate the traces and monitor the execution of the program. Here, Valgrind differs from Pin in the sense that Valgrind uses an architecture independent Intermediate Representation(IR) called 'Vex' which allows to write tools compatible with architectures that are supported by IR.

The way to extract software traces: A single trace of the whitebox binary with an arbitrary plaintext and a record of all accessed addresses and data is extracted. Here, the scope of the main executable is reduced to a library where cryptographic operations are happening. After that, a common computer security technique known as Address Space Layout Randomization(ASLR) is used which which arranges the address space position of the executables randomly, its data, heap, stack, and other elements such as libraries. After that, the trace is visualized to understand which block cipher is being used. This identification is done by understanding the repetitive patterns. For instance, if there are 10 patterns then the block cipher is for AES-128, 16 Round DES, 14 round AES-256, etc. Here, for visualization purposes, the trace was given a graphical representation. In the graph in Figure 10, the x-axis denotes the virtual address space such as stack address, heap, data segment, uninitialized data segment, etc. Also, the y-axis is nothing but a temporal axis here. In Figure 10, there are some significant colors. At the very beginning, black represents the instructions that are being executed. After that, green represents the memory location address that is being read. Finally, red represents the memory address locations that are being written. Once the algorithm is recognized, the trace is limited to a scope because usually in a large file the other portions might do some other type of instructions which may not be useful. Usually, the first or last round of the programme is considered to mount an attack on the input and output of the cipher. After the software traces of actual data or addresses from input or output are obtained, the bytes are serialized into vectors of 1s and 0s to observe them using DPA. However, the software setting used to integrate the programme is being worked with no measurement noise. Finally, using regular DPA tools the key is extracted.

Figure 5.2.2 shows a full software trace of AES-128 whitebox implementation. As stated above, the black, red, and green lines have their significance. On the very left, the loaded instructions are shown. As black lines have a density at the very beginning of the graph, it indicates some loops are used to execute a sequence of instructions repeatedly. However, if the graph is zoomed to an extent, ten repetitive patterns can be observed which indicates the programme is for (highly possible) AES-128 implementation. Moreover, from the graph, another observation can be made that, since the last round is shorter it omitted some instructions during the execution of the programme, and it is also similar to AES-128 implementation in the sense that MixColumn is omitted in the last round of the implementation.

As the whole programme is executing some loop of instructions, keeping the tracer limited to some specific memory range was not possible. However, after taking the full execution trace using DCA, bytes written to the stack(green lines) were taken and computed using DPA. As the programme was internally and externally encoded,

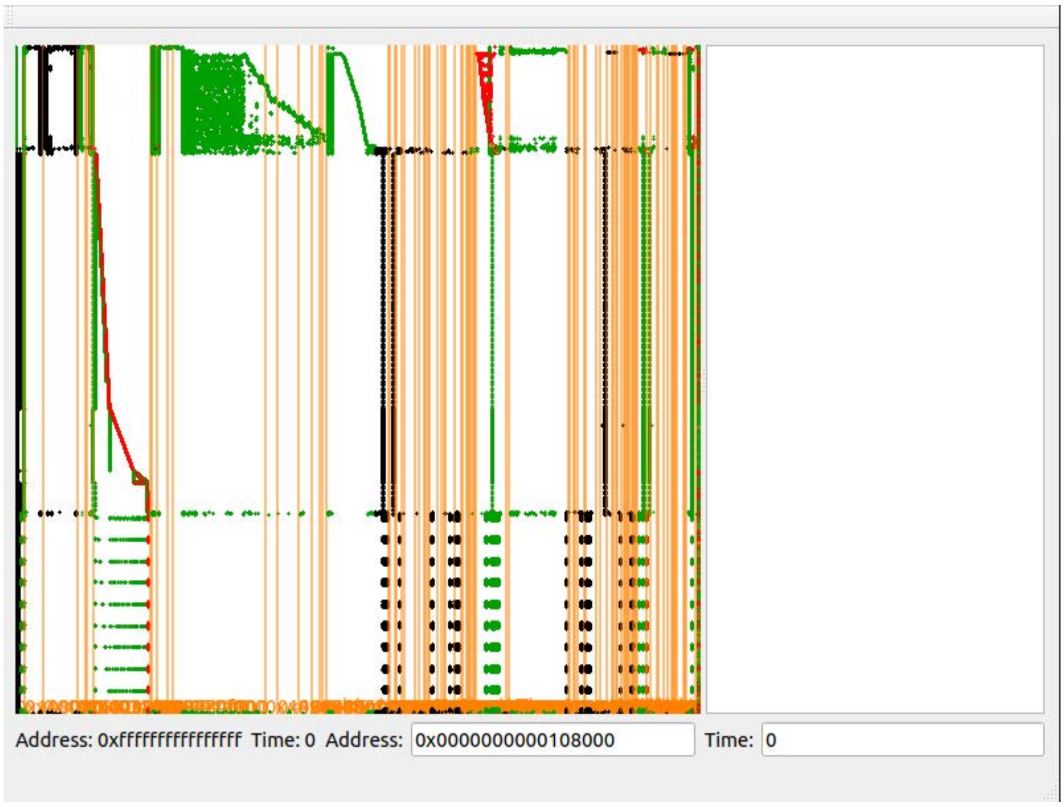

Address: 0xffffffffffffffff Time: 0 Address: 0x0000000000108000
Time: 0

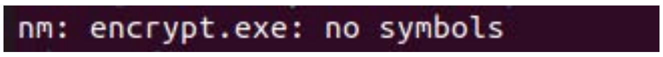
Figure 5.2.2: Graphical representation of AES whitebox implementation (ECB mode)

the intermediate states were not possible to observe directly. As a whole, the key extraction was not possible using DCA but the encryption algorithm was exposed. The whole process of tracing took less than an hour with some software and configuration management.

Symbol table stripping: The process where the symbol table information such as variables, functions, and other symbols used in a programme is removed or minimized from the final executable is known as symbol table stripping. Initially, this is one of the protective measures because after the symbol table is stripped the reverse engineering process becomes more difficult.

Reasons behind this difficulty are: The symbol table of a programme generally contains useful information such as functions' and variables' names which help reverse engineers to deeply understand the programme's logic. When this stripping is done, it becomes difficult for the attackers to extract information from the code. Moreover, Symbol table stripping is a part of code obfuscation. Without meaningful names, the attackers must rely on the usage of decompiled code which has no identifiers. Furthermore, debugging tools become ineffective after the stripping of symbol tables because debuggers use the information from symbol tables to convert machine code to high-level programming language.

For the process of symbol table stripping, a Linux-based tool known as 'strip' is used. The 'strip' command itself is a part of the GNU Binutils Package. After the usage of certain commands using the tool, the symbol table of the program will be deleted. As a whole, after the deletion of the symbol table, if another Linux-based tool ‘nm’ is run on the program, the following message will show:



```
nm: encrypt.exe: no symbols
```

Again behavior Integration of the graph: After the symbol table stripping, again the Tracegraph of the programme is generated. After the generation of the tracegraph, again the trace data investigation and behavior integration of the code is done. Consequently, the symbol table stripping made the reverse engineering of the code a bit difficult but the exposure of the programme remains like the previous one. Differential Computational analysis mainly focuses on the execution patterns of the programme rather than examining their debugging information or source code. As a result, symbol table stripping did not affect the observation that was obtained during the first analysis.

Behaviour Randomization of the graph(Implementation protected from DCA): Debugging tools can generate a memory allocation that provides a primary idea about the algorithm executing. To confuse the pattern, a random number of rounds can be executed in AES. For example, a random number of rounds, ranging from 11 to 20, can be selected for an AES algorithm implementation. Assuming the maximum number of rounds is set to 20, if the randomly generated number is ’r’, the total number of rounds becomes ’r’, with ’Garbage rounds’ defined as ’r-10’.

The most important thing to be kept in mind is that the goal is to confuse the memory allocation graph without changing the output cipher. To achieve this, the number of active rounds will remain unchanged from the original configuration. Additionally, it will store the latest state in a temporary array and restore the state just before the final round. As a result, the ultimate cipher remains unaffected by the additional garbage rounds.

Algorithm 7 Implement

```
1: if round is equal to 9 then
2:     for $i$ from 0 to a random round do
3:         temp[$i$] = state[$i$]
4:     end for
5: end if
6: for $i$ from 0 to a random round do
7:     state[$i$] = temp[$i$]
8: end for
```

After the extra randomization was added, again the behaviour of the programme was observed. The graph once showed 15 repetitive patterns and in another graph, it showed 13 rounds. As a whole, the programme achieved a certain bit of randomization. Consequently, an attacker who wants to apply different side-channel attacks or reverse engineering processes, or any other hacks will not have the clue to follow the trace of which algorithm, and altogether the algorithm of whitebox implementation will achieve a certain security perspective. Figure 5.2.3 shows the graph after randomizing 15 rounds and Figure 5.2.4 shows the graph after randomizing 13 rounds.

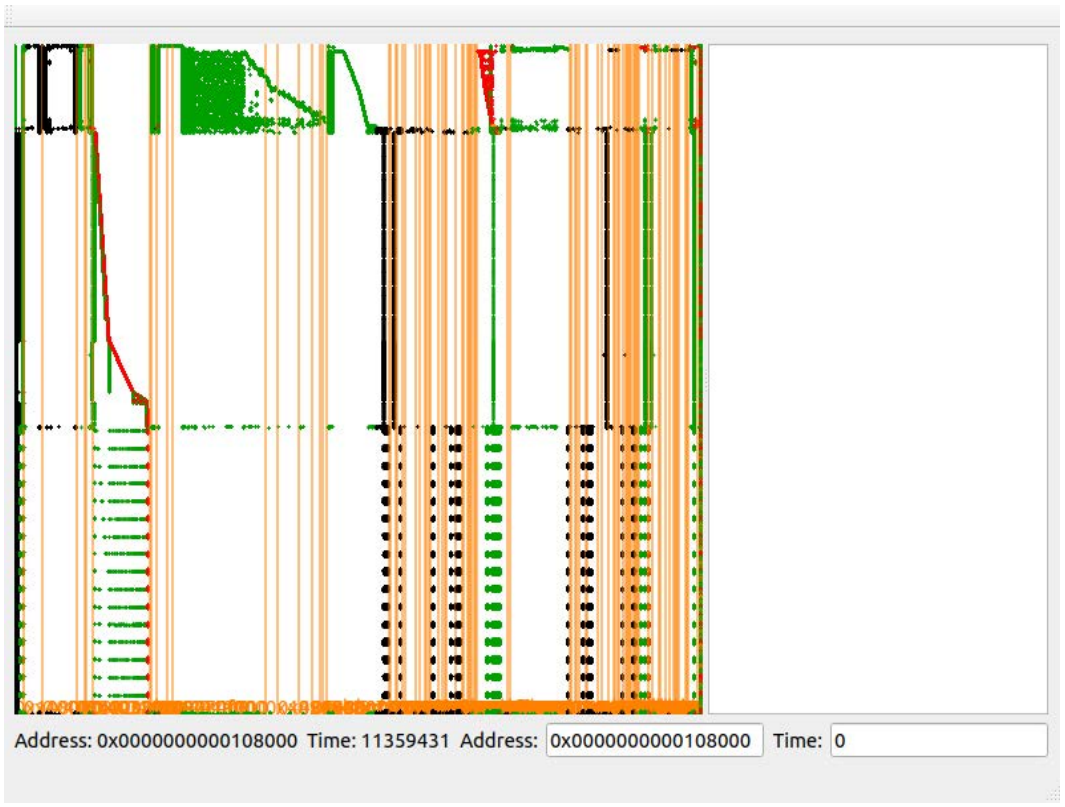
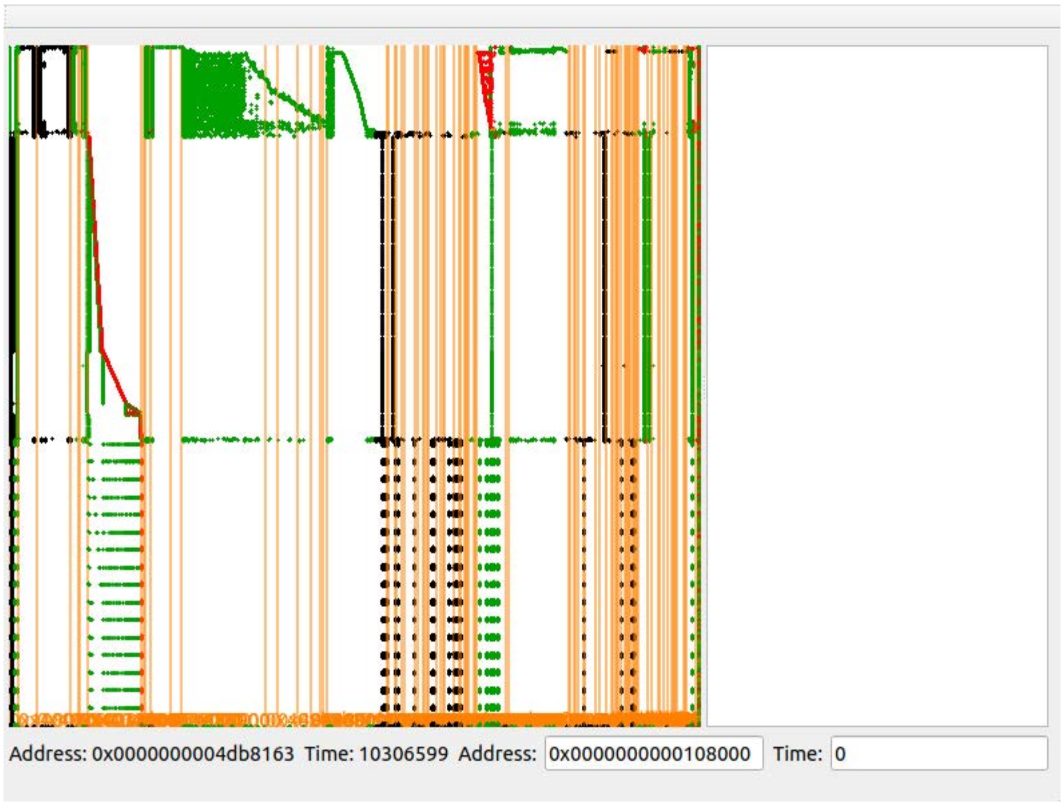

Address: 0x0000000000108000 Time: 11359431 Address: 0x0000000000108000 Time: 0

Figure 5.2.3: 15 rounds implementation



Address: 0x0000000004db8163 Time: 10306599 Address: 0x0000000000108000 Time: 0

Figure 5.2.4: 13 rounds implementation

A comparison of our implementation with the open source implementation of Choe et al in terms of different protective measures is given in the table below

Possible attacks	**Choe et al. Implementation**	**Our Implementation**
Reverse Engineering	The attacker has full access to the implementation system of that implementation. As a result, he/she can easily analyze the code, understand the binaries of the executable, and alter it. Moreover, the key extraction becomes possible with the help of the reverse engineering process.	As the symbol table of the executable is stripped, it is not possible to generate the symbol table and understand the internal architecture. That is why reverse engineering is not possible.
Code Lifting Protection	Code lifting protection, also known as code obfuscation, is described theoretically in the Choe et al. implementation. But the practical execution is not present in that.	The implementation has gone through three steps of code lifting protection. At first, the whole implementation was internally encoded. After that, the ciphertexts were externally encoded. Finally, randomization was added to the implementation to fortify the exposure of the algorithm.
Side Channel Attacks	The design of Choe et al's implementation was limited to the embedding of the key in the lookup tables. They were not protected by encoding. With the help of diffusion, the key could be extracted with a 2^{30} time complexity [dca paper].	Side channel attack depends on various factors such as the design of the implementation, the algorithm used for encryption, and the countermeasures applied to mitigate the vulnerabilities. In our implementation, the design is made considering the code obfuscation scenario, keeping in mind. Moreover, after adding randomization, the implementation surpassed Differential Computational Analysis. In the future, the vulnerability of the implementation will be checked with Differential Fault Analysis.

Comparison of previously Whitebox AES variants

Implementation	**Size**	**Best Attack**
Chow et al	773 KB	2^{22}
Karroumi	752 KB	2^{22}
Xiao - Lai	20 KB	2^{32}

5.3 Performance Analysis of Different Implementation Modes

The security of the Whitebox implementation was increased after adding randomization. However, the extra round or randomization will have an impact on the performance of the algorithm. As a consequence, the whole implementation was once again done in counter mode. After that, the implementation of both the ECB mode and CTR mode was analyzed. It was seen that Blackbox AES implementation of ECB mode takes the smallest runtime but it is insecure due to the exposure of the key. On the other hand, the Whitebox implementation of ECB mode after adding randomization takes much more time than the whitebox implementation of ECB without randomization. Comparing all the modes of whitebox implementation, Whitebox Counter mode (CTR) takes less time and also the performance of it does not get degraded after adding randomization. After randomizing, Figure 5.3.5 shows the performance analysis of different modes of Blackbox and Whitebox implementation.

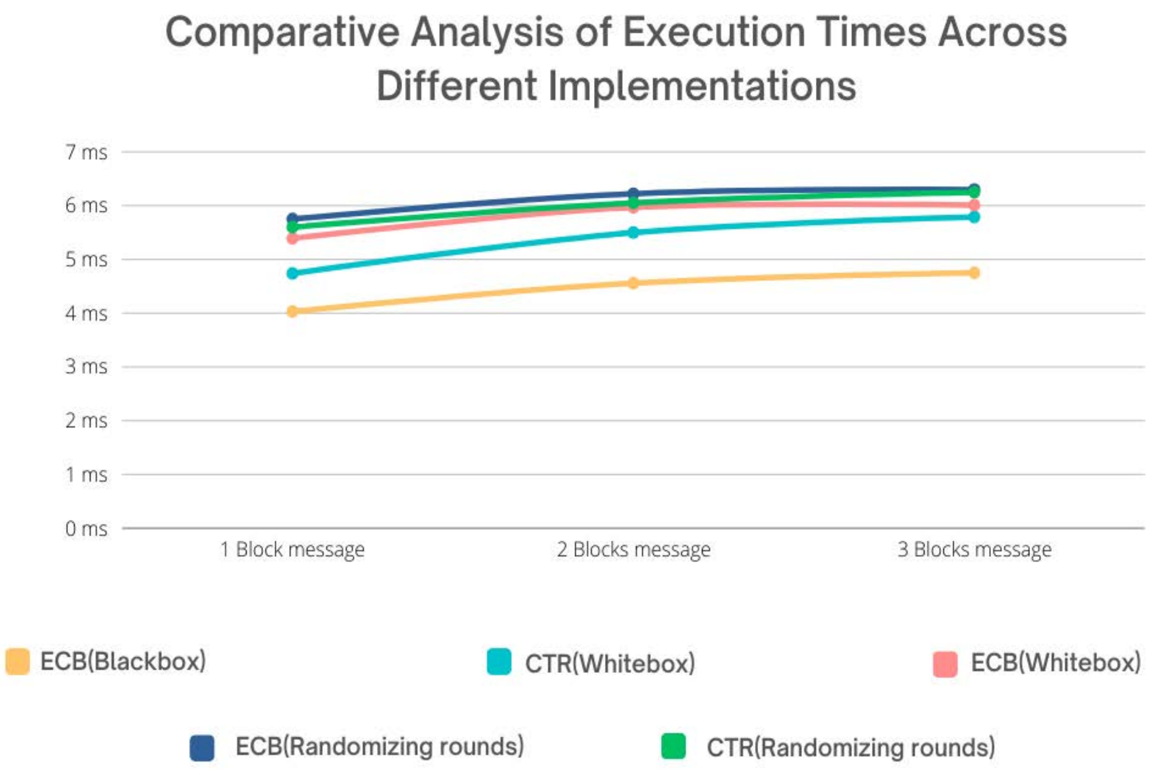


Figure 5.3.5: Performance analysis of different modes of Blackbox and Whitebox implementation after randomization

Chapter 6

Challenges

The challenge faced so far that we have encountered is that -

1. Whitebox related papers focus more on the theoretical discussion, no open source implementation of the algorithms is available.

2. The components that we have created are based on reading the papers and creating everything from scratch.

3. For the case of memory handling, it was not possible to do it in a language like Python that we all are used to but it was possible in CPP language, so we focused on Cpp language for its creation.

4. Other challenges were the implementation of the Whitebox, the conversion of the Blackbox to white box generation where the main issue was in the Whitebox after generating the table, loading it to the memory, and checking for performance. The performance of the Whitebox was slower compared to the Blackbox so here come the modes, ECB is the simplest mode but less secure whereas CTR has superior security with some performance trade-offs. We investigated different modes of the AES algorithm to make our algorithm more memory-efficient and CPU friendly.

5. Doing the Differential computational analysis (DCA), making trace from the execution and observing the behaviour of the code was quite challenging altogether.

Chapter 7

Conclusion

The field of Whitebox Cryptography is both a stimulating research problem and an essential component of the larger field of safe computing in untrusted contexts. The goal of this thesis was to explain the complex world of Whitebox cryptography while focusing on important ideas and adding to the continuing discussion in this area.

We began the journey by outlining the foundations of Blackbox cryptography, and its history, and then we began the implementation of Whitebox cryptography. While investigating the performance of different modes of the AES algorithm, we got that ECB is the simplest but least secure and memory consuming while CTR mode is a well-liked option when random access is required since it offers good security and high performance.

Afterward, the transformation technique started from the Blackbox cryptographic algorithm to the Whitebox algorithm by embedding the key into a series of lookup tables. In addition, this new lookup table-based implementation will also need protection which was done by applying internal and external encoding to the tables. At the last part of the paper, a comprehensive security and performance analysis was done to make the implementation more secure and memory efficient. However, designing effective countermeasures requires an understanding of the distinct risk model of White Box attackers who have unrestricted access to the algorithm's execution. Our investigation took us into the complex world of White Box schemes concentrating in particular on AES-based solutions.

It is clear from looking ahead that innovation is the key to achieving safe Whitebox cryptography. We must keep investigating fresh ideas, utilizing cutting edge primitives, and tackling the changing difficulties presented by algebraic attacks. Examining the S-box design and its resistance to cryptographic attacks suggests a viable direction for future study. Whitebox cryptography continues to be a difficult area in the constantly changing field of cybersecurity. We can defend our digital world against malicious attacks in untrusted contexts but only by thorough inquiry, innovation, and the constant search for safe solutions. Our adventure in Whitebox cryptography has just begun, and there is much potential to pioneer new solutions for a more secure digital future. This thesis concludes with a strong call to action [17].

7.1 Future Works

For advanced safety measurement, Differential fault analysis (DFA) will be applied to the implementation of the code and its efficiency against such sort of attacks will be checked. DFA is a kind of side-channel attack where an attacker makes introduction of faults such as voltage or clock glitches into the cryptographic system during its execution and analyzes the difference in the output. From the observation, the attacker gains information about the secret key.

Additionally, the implementation will be done using multithreaded concept. As of now, the implementation is based on single-thread system. Multiple threading will lower the computational power as the whole process will be divided into different cores. As a consequence, the performance measurement of the algorithm will be better. Moreover, the algorithm will be allowed to scale with the available hardware resources such as multicore processors and the responsiveness of the algorithm in real-time scenarios will be increased. As a whole, the resource utilization will be much better.

7.2 Project Links

For the code implementation, you can find the source code on GitHub:

https://github.com/rabib0/Implementation-of-WhiteBox-Cryptogtaphy.

For the Colab notebook, you can access it online:

https://colab.research.google.com/drive/1qiuyJjeQdWKHfIfoGGEQM4_D_MI4l4nm.

Bibliography

[1] M. Karroumi, "Protecting white-box aes with dual ciphers," Jan. 1970.

[2] W. Diffie and M. E. Hellman, "Privacy and authentication: An introduction to cryptography," vol. 67, no. 3, Mar. 1979.

[3] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," vol. 4, Aug. 1997. DOI: 10.1007/3-540-69053-0_4.

[4] S. Chow, P. A. Eisen, H. Johnson, and P. C. van Oorschot, "White-box cryptography and an aes implementation," Aug. 2002. DOI: 10.1007/3-540-36492-7_17.

[5] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, Jan. 2002, p. 238, ISBN: 3-540-42580-2. DOI: 10.1007/978-3-662-04722-4.

[6] A. Young and M. Yung, "The dark side of "black-box" cryptography or: Should we trust capstone?," 2002.

[7] P. Dusart, G. Letourneux, and O. Vivolo, "Differential fault analysis on a.e.s," *LNCS*, vol. 2846, Oct. 2003.

[8] O. Billet, H. Gilbert, and C. Ech-Chatbi, "Cryptanalysis of a white box aes implementation," Jan. 2005. DOI: 10.1007/978-3-540-30564-4_16.

[9] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," vol. 1294, Jan. 2006.

[10] X. Lai, "A secure implementation of white-box aes," Dec. 2009.

[11] J. Persial, P. M, and D. R, "Side channel attack-survey," 2011.

[12] C. J. Benvenuto, "Galois field in cryptography," May 2012.

[13] sysk, "Practical cracking of white-box implementations," Apr. 2012.

[14] T. Lepoint, M. Rivain, Y. D. Mulder, and P. Roelse, "Two attacks on a white-box aes implementation?," Aug. 2013. DOI: 10.1007/978-3-662-43414-7_14.

[15] J. A. Muir, "A tutorial on white-box aes," 2013. DOI: 10.1007/978-3-642-30904-5_9.

[16] A. " Souchet, "Aes whitebox unboxing: No such problem," 15 2013.

[17] J. W. Bos, C. Hubain, W. Michiels, and P. Teuwen, "Differential computation analysis: Hiding your white-box designs is not enough," Aug. 2016. DOI: 10.1007/978-3-662-53140-2_11.

[18] J. Cho, K. Y. Choi, O. Dunkelman, and N. Keller, "Hybrid wbc: Secure and efficient white-box encryption schemes," Oct. 2016.

[19] J. Bhatia, "Comparison of white box, black box and gray box cryptography," 2017. [Online]. Available: http://dx.doi.org/10.21172/ijiet.82.031.

[20] E. Alpirez Bock, J. W. Bos, C. Brzuska, *et al.*, "White-box cryptography: Don't forget about grey-box attacks," *J. Cryptol.*, vol. 32, no. 4, pp. 1095–1143, Oct. 2019, ISSN: 0933-2790. DOI: 10.1007/s00145-019-09315-1. [Online]. Available: https://doi.org/10.1007/s00145-019-09315-1.

[21] X. LOU, T. Zhang, J. Jiang, and Y. Zhang, “A survey of microarchitectural side-channel vulnerabilities, attacks and defenses in cryptography,” Mar. 2021.

[22] Baboon, *HackLu Reverse Challenge - Baboon’s Blog*, http://baboon.rce.free.fr/index.php?post/2009/11/20/HackLu-Reverse-Challenge, [Accessed 07-01-2024].

[23] W. Brecht, “White-box cryptography: Hiding keys in software,”

[24] *NoSuchCon 2013 challenge - Write up and Methodology*, https://kutioo.blogspot.com/2013/05/nosuchcon-2013-challenge-write-up-and.html, [Accessed 07-01-2024].