8-24-2016

An Efficient Leakage Free Countermeasure of AES against Side Channel Attacks

Abdulaziz M. Miyajan
University of Connecticut, abdulaziz.miyajan@uconn.edu

An Efficient Leakage Free Countermeasure of AES against Side Channel Attacks

Abdulaziz Miyajan, Ph.D

University of Connecticut, 2016

ABSTRACT

Across generations many electronic devices have become more popular starting from cell phones, tablets, laptops, desktops and any communication devices. Globalization and various treaties have resulted in increased outsourcing of data, software, hardware and various services. This increases the need to protect data and information whether transmitted, processed, stored or distributed, which results in developing many cryptographic algorithms to secure different types of hardware and software. In 2001, the National Institute of Standards and Technologies (NIST) selected Rijndael as the advanced encryption standard (AES) [1]. The security of AES has received considerable attention, which encourages the community to compete in developing many hardware and software implementations for AES. AES has become the widely used encryption primitive to protect electronic data in many applications and platforms. Side-channel attacks (SCAs) are a real and successful threat to AES. SCAs obtain secret keys by exploiting the physical leakage of information from executing cryptographic algorithms such as those for power consumption, electromagnetic radiation, running time and cache profile information. In power and electromagnetic analysis attacks, the observable leakages are the power consumption and the electromagnetic radiation of the device, which are dependent on the processed data and the performed operations. Masking is a common and widely used countermeasure to defeat differential power analysis (DPA) and differential electromagnetic analysis (DEMA) attacks. In timing attacks which can be carried out remotely (e.g. over networks [2]), the observable leakage is the execution time variation to respond to queries or perform cryptographic operations that depend on critical data or secret keys. Having a constant-time implementation is a sound countermeasure to mitigate timing attacks. In cache attacks, an observable leakage (covert channel) about which part of the look-up table was accessed, might be revealed by the cache. One way to mitigate cache attacks is by having an implementation that does not use lookup tables. Another way is by using Data-Oblivious Memory Access Pattern. The second technique requires a small increment of memory but is more efficient, while the first one solves the memory overhead problem with negative impact on the performance. This thesis proposes an efficient, leakage-free countermeasure of AES, against multiple side channel attacks, meanwhile avoiding as much as we can, the negative impact in the performance as a result of combining those countermeasures, and the memory overhead that results from storing lookup tables.

The first part of the thesis focuses on techniques to produce an efficient leakage-free countermeasure of AES against power analysis, electromagnetic emissions, timing and cache attacks. This countermeasure features a secure higher-order masking scheme (to defeat power and electromagnetic attacks), the elimination of lookup tables (to mitigate cache attacks and remove memory overhead), and a constant-time implementation (to defend against timing attacks). However, combining these countermeasures imposes a negative impact on performance. Therefore, we propose techniques to solve the performance problem at the algorithm and data levels.

The second part of the thesis focuses on the bottleneck transformation of the AES which is the SubByte transformation. This thesis demonstrates the techniques, presenting an application of efficient SubByte transformation by working in three different levels: algorithm, byte and bit levels.

The last part of the thesis focuses on the inversion operation which is the first part of the SubByte transformation, utilizing the squaring property of the normal basis. Consequently, these masks need to be converted between the two forms (polynomial and normal basis) based on the sequence of operations. This thesis presents a low cost basis conversion technique that can be applied with or without using lookup tables. Moreover, this thesis presents efficient and secure techniques, which mitigate cache attacks and further improve the running time of the algorithm, in exchange for a small memory requirement.

Finally, to verify our work we deployed the several techniques in a real application (OpenSSL framework) as a case study.

An Efficient Leakage Free Countermeasure of AES against Side Channel Attacks

Abdulaziz Miyajan

M.S., University of Connecticut, 2012

B.S., Um Al-Qura University, 2005

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

At the

University of Connecticut

2016

APPROVAL PAGE

Doctor of Philosophy Dissertation

An Efficient Leakage Free Countermeasure of AES against Side Channel Attacks

Presented by

Abdulaziz Miyajan, B.Sc., M.Sc.

Major Advisor ____

Chun-Hsi Huang

Associate Advisor ____

Sanguthevar Rajasekaran

Associate Advisor ____

Reda A. Ammar

University of Connecticut

2016

ACKNOWLEDGEMENTS

All thanks and praises are due to Allah, the creator of the creations. As part of his blessing, generosity, grace and mercies on me, I got the power, support, help and the ability to complete my thesis. This thesis is the result of many years of research which see the light by the supervision and help of many people starting from my previous advisor Prof. Jerry Shi, who started this work with me. Then special thanks to my great advisor Prof. Chun-Hsi (Vincent) Huang, the professor and the friend who accepted me after the leave of Prof. Jerry Shi and advise me through my career. In addition, I am very grateful to my great associative advisor, Prof. Turki F. Al-Somani, the professor, the brother and friend who helped me a lot with his time, knowledge and funds for my research and my life. Here, I would like to thank Prof. Reda Ammar, the professor and the father who loves to help others. Moreover, I would like to thank all the Uconn CSE professors who I learned from. I hope all the best for my parents, who always pray for me to success in my life. I extend my deepest thank to my family, my wife and kids for the patient and support during my career. Furthermore, this succeed would be impossible without the great scholarship from Umm Al-qura University which located in Makkah, the city where I was born and raised and work, in my great and generous country Kingdom of Saudi Arabia that does not hesitate to help educating our people.

DEDICATION

To my God

To my parents

To my family (Raghda, Mohammed, Layan and Yara)

Table of Contents

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 History and motivations

As electronic devices become more popular, the need to protect transmitted, processed, stored and distributed data increases. The security of different types of hardware and software has become a major and critical concern, and has resulted in the development of cryptographic algorithms. Since 2001, when the *National Institute of Standards and Technologies* (NIST) selected *Rijndael* as the advanced encryption standard (AES) [3], many hardware and software implementations of AES have been proposed and it has become the most popularly used encryption primitive. Many applications in different platforms, from high-end servers to mobile consumer products, use AES as the encryption primitive to protect their electronic data. Network communication protocols that use AES include TSL, VPNs, Wi-Fi, MTP and SSL. Automated teller machines (ATMs) use AES to securely transmit sensitive information to their information processing centers [4]. Cloud computing is being used for applications such as social networking, online document services, email, backup, secure multiparty computation (SMC), banking, and financial services, which are retrieving and manipulating previously stored data in the Cloud. For privacy and integrity reasons, AES is used to encrypt data before storage and to decrypt the data when retrieving them. Hackers' attention is therefore focused on AES. Side-channel attacks (SCAs) are a real and successful threat to AES. SCAs obtain secret keys by exploiting the physical leakage of information from executing cryptographic algorithms such as those for *power consumption*, *electromagnetic radiation*, *timing variations* and *cache profile information* [5] [6] [7] [8] [9].

Extensive efforts have been made to propose countermeasures against SCAs at the hardware level. However, less has been proposed at the software level, which is also critical and subject to improvement. Cache attacks, and many design techniques for their mitigation are discussed in [9] [10]. One way to mitigate cache attacks is by having an implementation that does not use lookup tables. Another way is the Data-Oblivious Memory Access Pattern that will be discussed later. Timing attacks are made possible by visibility of the variability of execution time of a code due to conditional branches that depend on critical data or secret keys [7]. Indeed, developing a countermeasure to mitigate timing attacks can be achieved by having a constant-time implementation of the protected algorithm [11]. One benefit from using SIMD technology in our work is to obtain a constant-time implementation. Differential power analysis (DPA) and differential electromagnetic analysis (DEMA) [6] are two different types of serious SCAs that are non-invasive and can be mounted without any knowledge about the targeted device; the first one is targeting the power consumption while the later one is targeting the electromagnetic radiation; both have to use the same statistical techniques to explore some important information to obtain the secret key. These types of attacks cannot be detected by a device because they are non-invasive and passive, exploiting externally available information using cheap equipment without direct access to the interior of the target device. As a result, traditional attack detectors are ineffective and the countermeasure designers will need to ensure that power consumption of the device does not correlate with the sensitive information being processed. *Hiding* and *masking* are two well-investigated solutions at the algorithmic level, where the first one makes the leakage constant while the other one makes the leakage dependent on some random value. In this thesis, we will focus on the masking strategy.

Masking is a common and widely employed countermeasure to protect block cipher implementations against SCAs especially at the software level. In masking, for every execution of the algorithm, the input data and the secret key (sensitive intermediate variables) are obscured with fresh and randomly selected bits. Therefore, all the computations at the algorithm level are masked until the end of the last round to break the correlation between the secret key and the actual power consumption. The final results are unmasked to retrieve the correct results. Several first order masking schemes are proposed in [12]. Due to this, adversaries generalized the principle, exploiting several leakage points at the same time; this is often referred to as higher-order SCA. To mitigate this type of attack higher-order masking countermeasures are proposed but their execution consumes a significant amount of time. Cryptosystem engineers have since been focusing on designing efficient higher-order masking countermeasures [12] [13]. The first provably secure higher-order masking scheme was proposed in [14].

In [14], the OpenSSL, which is an open source library collection of many cryptographic protocols [15], implementation of the AES algorithm was exploited by electromagnetic SCA using an Intel Atom processor. Furthermore, in [16] the OpenSSL implementation of the AES algorithm was exploited by a cache timing SCA using a Pentium III processor. Moreover, in [8], electromagnetic attacks were successfully mounted on modern laptop and desktop systems with various Intel processors.

Unfortunately, applying one countermeasure against one of the SCAs can mitigate that type but might leave the cryptographic implementation vulnerable to another type of attack. The above issues inspired us to produce a combined countermeasure to mitigate different types of SCA simultaneously. However, applying each countermeasure will cause a serious impact on the performance or memory requirements. Remarkably, we face three challenges, the security of AES against multiple SCAs at the same time, the performance impact and the memory overhead to store lookup tables. These challenges can be overcome through the development of a combined countermeasure that will secure AES from those attacks while including techniques to improve the performance and solve the memory overhead issue. This can be achieved by combining a provably secure higher-order masking scheme (to defeat power and electromagnetic attacks), and a constant-time implementation (to defend against timing attacks), with either the elimination the use of lookup tables or use of Data-Oblivious Memory Access Pattern (to mitigate cache attacks and remove or reduce memory overhead), followed by speed up techniques on algorithm, byte and bit level.

The thesis focuses on the implementation of various beneficial aspects of cryptography, more specifically how to properly integrate multiple security techniques into one solution, so that the implementation not only greatly increases the security, but also in an efficient manner. Although algorithmic security has reached a proven degree of success in providing security, obtaining physical security of a cryptographic implementation while maintaining an acceptable speed, is much more difficult to accomplish. Due to the increasing demand for the address of flaws in implementation, cryptographic engineering, a new discipline has rapidly developed, which focuses specifically on security and efficiency of cryptographic implementations.

1.2 Cryptography

Cryptography is the art and science of using mathematics to produce protocols, algorithms and techniques to hide and secure sensitive information in transit in the presence of adversaries. (Originally, it was focused only on the problem of secret communication.) Thus, securing the sensitive information (called *plaintext*), requires converting it into indecipherable format (*ciphertext*) using a special secure process (called *encryption*) before accessing over an untrusted medium (e.g., Internet), and converting it back (by the authorized recipient), into the original plaintext using a special secure process (called *decryption*) when retrieving it over that untrusted medium. Even if an adversary succeeds in accessing the ciphertext, he or she will not be able to recover the plaintext from it. Combining those encryption and decryption algorithms introduces a new terminology called *cipher*.

In the late 20th century, a new terminology (called *modern cryptography*) was raised with the invention of computers, cell smart phones, tablets and other communication devices to solve more problems than *confidentiality* such as *authentication*, *Integrity*, *exchanging secret keys*, *non-repudiation*, etc. Commonly, modern cryptography focuses on:

- **Confidentiality**: the intended recipient is the only one who can decrypt the ciphertext to extract the original plaintext.
- **Authentication**: the intended recipient is able to verify the identity of the sender from the received message.
- **Integrity**: the intended recipient is able to verify that the message has not been changed during the transition process from the received message.
- **Non-repudiation**: to ensure that the sender cannot deny sending the message later.

In [17], they define modern cryptography to be "The scientific study of techniques for securing digital information, transactions, and distributed computations". Today, cryptographic algorithms are integrated in every computer system. Cryptographic algorithms are used in many applications such as enforcing access control in multi-user operating systems, preventing extracting secrets from stolen devices, and preventing copying, etc. Thus, cryptography has gone from strictly securing communication to helping secure systems everywhere, to receiving intense study within computer science. In modern cryptography, the attacker is assumed to be aware of the algorithm but not the cryptographic key, which is used by the algorithm to determine the output. Cryptographic

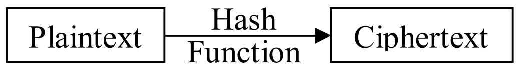


Figure 1 Hash Function

algorithms can be classified based on the number of keys employed for encryption and decryption into three types:

1. **Hash Functions:** Does not use any key since it is a one way mapping and unrecoverable as shown in Fig.1.

   Hash Functions: also called *message digests* are an efficient one way mapping

that takes the entire message as input (no secret key required) to compute a short fixed length string called hash value, in which the plaintext is non-reversible by any means. Collisions (where two input messages produce the same hash value using the same hash function), to break the hash function, are excessively difficult to find. In the security field, hash functions widely provide data integrity, digital fingerprints, password encryption etc. Examples of commonly used hash algorithms include Secure Hash Algorithm [18] (SHA-1 [19], SHA-2 [18], and SHA-3), Message Digest (MD) algorithms (MD2 [20], MD4 [21], and MD5 [22]), RIPEMD [23], etc.

2. **Secret (Symmetric) Key Cryptography (SKC):** Uses a single key for both encryption and decryption as shown in Fig.2.

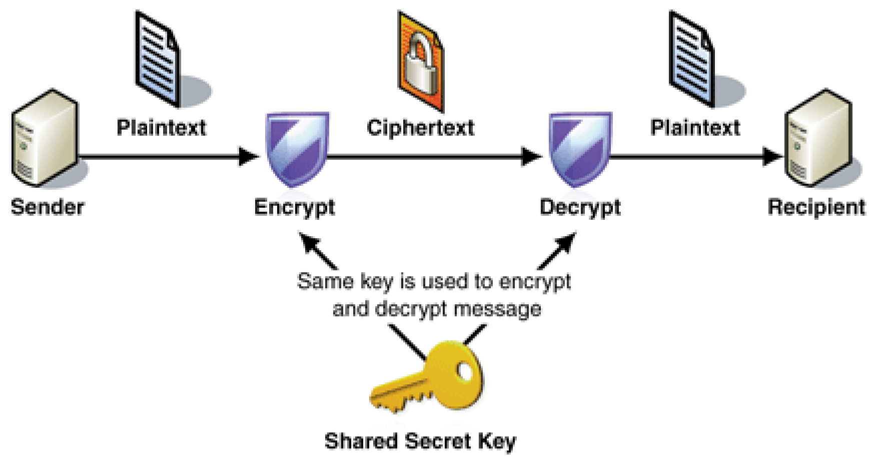


Figure 2: Symmetric Key Cryptography [24]

Secret Key Cryptography: also called *symmetric encryption* is another type of cryptographic algorithm where the sender and the receiver must use the same secret key for encrypting the plaintext and decrypting the ciphertext. For security purposes, the secret key must be known for both parties (the sender and the receiver) before starting the communication, which makes the key distribution between both parties the most difficult part. In general, secret or Symmetric Key cryptography are classified into *stream ciphers* or *block ciphers*.

Stream ciphers encrypt the plaintext by combining single bits, bytes, or computer words, at a time with a continuously changed key stream, using the exclusive-or (Xor) operation. Stream ciphers apply some form of feedback mechanism to create a stream of key material with arbitrary length that is equal to the length of the message, moreover ensuring that the key stream is continuously changed. Stream ciphers, when combine the plaintext bit-by-bit, byte-by-byte or character-by-character with the key stream, will behave similarly to the one-time pad. In a stream cipher, the output stream is generated

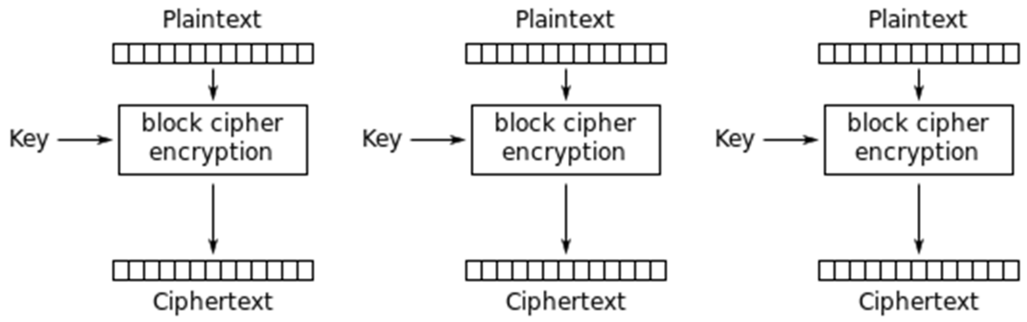


Figure 3: Electronic Codebook (ECB) mode encryption [52]

based on the internal state of the cipher, which changes according to how the cipher is designed. That state change in stream ciphers is manipulated either by the key, or in some instances by the stream of plaintext [24]. Stream ciphers come in several types but self-synchronizing stream ciphers and synchronous stream ciphers, in particular, have much more importance in application than the others. Examples of commonly used and well known stream cipher are Rivest Ciphers (RC1, RC2 [25], RC3, RC4 [26], RC5 and RC6 [27]).

A block cipher [28], in comparison to a stream cipher, is named based on how it functions. After dividing the plaintext into blocks based on the acceptable block length of the used cipher, the same secret key will be applied to encrypt each block in the scheme. Therefore, encrypting multiple identical plaintext blocks will always correspond to the same ciphertext in a block cipher, however, in a stream cipher, using the same plaintext will encrypt to different ciphertext. Examples of block ciphers include DES (Data Encryption Standard [29] [30]) and AES (Advanced Encryption Standard [3] [31] [32]).

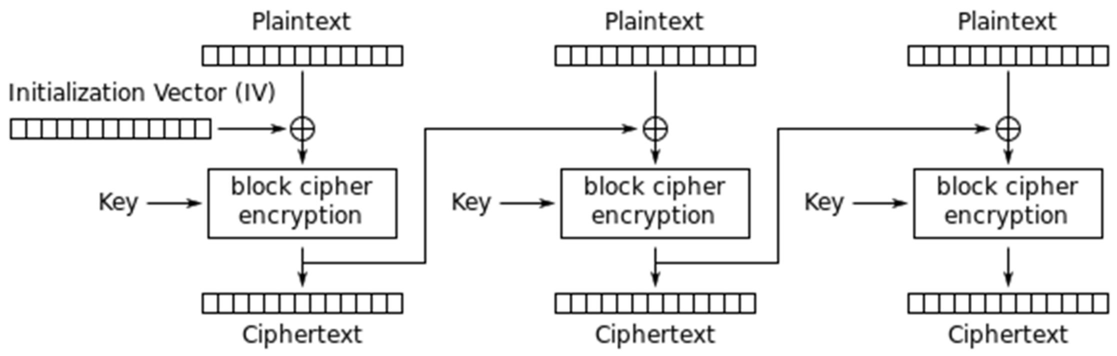


Figure 4: Cipher Block Chaining (CBC) mode encryption

There are several modes of operation that have been recommended in [33] for symmetric key block ciphers, the most important of which are Electronic Codebook, Cipher Block Chaining, Cipher Feedback, and Output Feedback.

- **Electronic Codebook (ECB) mode** is the least complex, most common, and most apparent in applications. An ECB mode uses the same secret key to encrypt each individual block of the plaintext to produce a ciphertext block, which makes it easy to parallelize the encryption process using multiple processors as shown in Figure 3: Electronic Codebook (ECB) mode encryption and makes this mode prone to brute force attacks. Therefore any two identical plaintext blocks will generate the same ciphertext block when using the secret key.
- **Cipher Block Chaining (CBC) mode** is the most complex mode and functions by adding a feedback mechanism to the encryption scheme. In CBC, the plaintext is exclusively-ORed (XORed) with the ciphertext block before it, prior to encryption as shown in Figure 4. This mode is impossible to parallelize because each input for each block is dependent on the output of the block before it.

Therefore any two identical plaintext blocks will never generate the same ciphertext block when using the secret key.

- **Cipher Feedback (CFB) mode** is also a block cipher implementation, however, it is a self-synchronizing cipher. CFB mode delays Xoring plaintext compared to CBC. Both permit data to be encrypted in units smaller than the block size, (such as byte-by byte) which might be useful in some applications such as encrypting interactive terminal input. In 1-byte CFB mode, for example, each incoming byte is placed into a shift register, encrypted bit by bit, and the block transmitted. At the receiving side, the ciphertext is decrypted and the extra bits in the block (i.e., everything above and beyond the one byte) are discarded.
- **Output Feedback (OFB) mode** is a block cipher implementation that uses a weaker tap location for feedback, and is to a synchronous stream cipher. This feedback prevents the same plaintext block from generating the same ciphertext block while being independent of both the plaintext and ciphertext bitstreams.

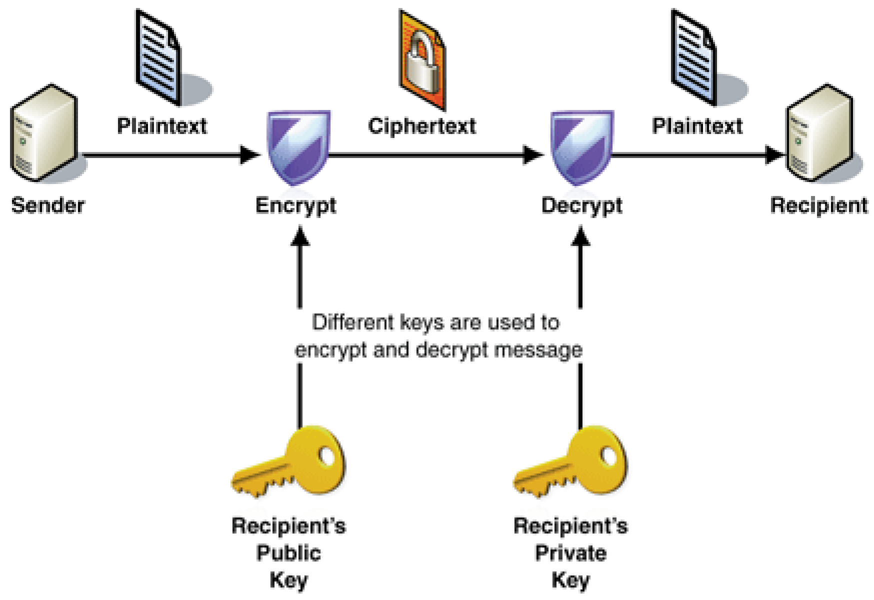


Figure 5 Public Key Cryptography [35]

3. **Public Key Cryptography (PKC):** Uses two keys, one for encryption and another for decryption.

Public key cryptography: also called *asymmetric key cryptography* is another type of cryptographic algorithms where the sender uses the recipient's public key to encrypt the plaintext and the recipient uses his own private key to decrypt the ciphertext and extract the original message as shown in Figure 5. PKC is one of the greatest developments in cryptography in hundreds of years. Public key cryptography relies on the concept of one way functions in order for two parties to engage in secure communication without the use of the same key, over non-secure channels. One way functions are mathematical functions that are easy for a computer to solve, while the inverse of the same function is anticipated to be difficult. For example, two numbers can be multiplied to get a product, but factoring that product to get two numbers can be difficult as many pairs of factors can be used to get a product. The same applies to logarithmic functions as the user can choose what number they would like to raise to a desired power. However, finding the two numbers using the inverse would have many possibilities and take a considerable amount of time to solve. There are two types of public key algorithms:

- **Encryption Algorithms**: this type uses two different keys (public key to encrypt and private key to decrypt) to solve the confidentiality problem. The most commonly used public key encryption algorithm today is RSA [34].
- **Digital signature algorithms**: this type uses two different keys (private key to sign the message and the public key to verify the signature) to solve the authentication problem. The most commonly used digital signature algorithm today is DSA [35].

1.3 Advance Encryption standard (AES)

In the 1970's, IBM developed Data Encryption Standard (DES), which later was broken by the supercomputer in less than 24 hours. In 2001, a new sophisticated algorithm called Rijndael, proposed by Vincent Rijmen and John Daemon, was selected by the National Institute of Standards and Technology (NIST) to be the advanced encryption standard (AES).

AES is a symmetric key block cipher that must use the same secret key for encryption and decryption [19]. AES has a mathematical formulation in the field GF(2^8), where the addition is the XOR operation (denoted by $\oplus$) and the multiplication (denoted by $\bullet$) of polynomials modulo the irreducible polynomial n(x) = $x^8 + x^4 + x^3 + x + 1$; (0x11B in hexadecimal representation). Before AES begins encryption, the data must first be arranged in a two-dimensional (4 rows by 4 columns) matrix of bytes called the state matrix. The AES algorithm has to satisfy the properties of key mixing, substitution, and linear transformation layers in the evaluation of all of its n rounds to produce the cipher text. The number of rounds n is slightly different based on the key length. There are three key lengths that are suitable with AES (128, 192, and 256); the corresponding numbers of rounds are 10, 12 and 14 respectively. Hence each AES round consists of four major transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The last round, however, avoids the MixColumns transformation [1]. In the SubBytes transformation, the state matrix bytes are replaced using Rijndael's S-Box in case the whole lookup table is stored. On the other hand, it can be calculated for each element of the state matrix by applying two operations, the multiplicative inverse using the power function $x \rightarrow x^{254}$ over the Galois field GF(2^8) followed by the affine transformation to achieve the substitution layer role. ShiftRows is the simplest transformation; the four rows are cyclically shifted left by n-1 bytes; so, zero, one, two and three respectively. MixColumns is a column-wise linear transformation rather than row-wise as in the previous one. It multiplies each column by a fixed 4×4 matrix as in (5). The linear layer role is satisfied with these two transformations. In AddRoundKey, each byte in the state matrix is XORed with the corresponding round key to meet the key mixing layer role.

1.4 Cryptanalysis

The objective of a cryptanalyst, someone who conducts cryptanalysis, is to figure out the breakdown of a cryptographic algorithm (either from the mathematical or the implementation side) in order to find the secret key, that helps decrypt all previous and new messages using that secret key. The process of determining the secret key of a cryptographic algorithm is known as an *attack*. A *brute force attack* is when an attacker performs a search on all the possible key candidates, and in the process determines the correct key. Moreover, if the attacker is able to find any fragility in the algorithm that reduces the complication of finding the secret key compared with the brute force technique, this attack succeeds. The goal of the cipher is to make the key and the encrypted message secure and prevent a brute force attack. In cryptanalysis, the details of the cryptographic algorithm are usually known to the attackers. Hence, based on the type and amount of information available, cryptanalytic attackers are classified into many types including:

- **Ciphertext only attack**: is the case when the attacker has control or access to multiple encrypted (ciphertext) messages using the same secret key.
- **Known plaintext attack**: is the case when the attacker has control or access to multiple messages (plaintext) and the produced encrypted (ciphertext) messages.
- **Chosen plaintext attack**: is the case when the attacker has control or access to encryption device. So, the attacker can encrypt multiple messages (plaintext) of his own choice and can produce the equivalent encrypted (ciphertext) messages.
- **Chosen ciphertext attack**: is the case when the attacker has control or access to encryption device. So, the attacker can decrypt multiple encrypted (ciphertext) messages of his own choice and obtain the equivalent decrypted (plaintext) messages.

- **Side channel attacks**: is the case when the attacker can utilize the side channel information of running the cryptographic algorithms on any device such as power consumption, electromagnetic radiation, timing variations etc.

1.5 Implementation or Physical Attacks

In the literature cryptographic primitive is considered from two points of views: classical or traditional cryptanalysis and the physical security. Classical cryptanalysis exploits the mathematical weaknesses in the algorithm to find the secret key, while implementation or physical attacks exploits the physical leakage information from running the primitive cryptographic algorithm on electronic devices that present a specific characteristics to recover the secret key involved in the computations. In fact, the security of the cryptographic algorithm is very important, but the security of the whole system is much more important to be considered, i.e the cryptographic device that executes the cryptographic algorithm. Thus, *cryptographic devices* as defined in [36] in page 3 “cryptographic devices are electronic devices that implement cryptographic algorithms and store cryptographic keys”. Cryptographic devices manufactures take into consideration the implementation attacks seriously because they are much more powerful and dangerous than the traditional attacks from their point of view.

In the literature implementation or physical attacks are classified based on two criteria:

- **Invasive vs. non-invasive attacks**: in the invasive attacks the attacker can control the cryptographic device, where he can depackage the chip, in order to observe the behavior or modify the functionality of the chip. It is the strongest and most costly type of physical attacks. In the non-invasive attacks the attacker can only observe the external phenomena of the cryptographic device. It is inexpensive.

- **Active vs. Passive attacks**: in the active attack the attacker tries to manipulate the functionality of the cryptographic device, aiming to induce abnormal behavior that results errors in the computations, to recover the secret key. As an example of this type of attacks include fault attacks [37] [38]. In the passive attacks the attacker utilizes the observable phenomena (leakage of information) from the physical implementation of the cryptographic algorithm during the execution. This type of attacks include power analysis attacks, which utilize the power consumption of the cryptographic device while running a certain operation depends on the secret key to collect some information about that key in order to recover it as in [36]. Second classes are timing attacks, which utilize the execution time variation of the same operation or algorithm for different inputs [7] [39]. Third classes are electromagnetic attacks, which utilize the correlation between the electromagnetic emanations produced by running the cryptographic algorithm and the secret key [40] [14].

1.6 Side Channel Attacks

Side channel attacks are a class of implementation attacks, where the attacker obtains secret keys by exploiting the physical leakage of information from executing cryptographic algorithms such as those from power consumption, electromagnetic radiation, timing variation and cache profile information. Side channel attacks are non-invasive, passive and are very easy to mount using inexpensive equipment. In practice, they are a serious threat to cryptographic devices starting from high-end servers such as dedicated cryptographic servers to small embedded devices such as smart cards. Side channel attacks were first introduced by Paul Kocher in 1996 [7]. The most successfully mounted side channel attacks include power analysis attacks [6], electromagnetic attacks [40], timing attacks [7] and cache attacks [41].

- **Timing attacks**. Cryptographic algorithms consume different amounts of time to process different inputs, due to the data dependent (non-fixed) running time of different operations [42]. For example, in the square-and-multiply method, if the square operation is performed on a 0, there are no operations, vs. if the bit is 1, then one square and multiply operation is performed. Those time variation measurements from any vulnerable system are utilized by the attacker to recover the secret key. In this type of attack the attacker assumed to know the design of the cryptographic system.
- **Cache attacks**. In modern CPUs that use the structure of memory caches, a cross-process information leakage is feasible as a result of the indirect interaction between those running processes sharing a processor. The cache works as a shared resource for all processes where it affects and gets affected by each process. Virtual memory mechanisms are used to protect the data stored in the cache, while the metadata and memory access patterns are not fully protected. Attackers can utilize the leakage information about the memory access patterns of another process, which affect the state of the cache (during or after encryption)

and the running time of the encryption. Similar to timing attacks, attackers can utilize the time it takes for cache hit/miss, to perform cache-timing attacks. Another way is, if the attacker is able to find which part of the lookup table was recently visited or the memory access patterns. Cache attacks include two families of attacks; Synchronous Known-Data Attacks and Asynchronous Attacks. For more details refer to [41].

- **Power analysis attacks**. A power analysis attack is one of the most effective side channel attacks. The power consumed by a circuit in any electronic device varies based on transistor activity and other components. In fact, the electronic device's power consumption includes information about the processed data and the performed cryptographic operations. The attacker can measure the power consumption by monitoring the power supply of the system while it is performing any cryptographic operation. This relationship, between the power consumption and the internal state of a cryptographic implementation, can be exploited by the adversary to recover the secret key. Refer to [36] for more detailed about power analysis attacks.
- **Electromagnetic attacks**. Electromagnetic attacks are exactly the same as the power analysis attacks except they target the electromagnetic radiation of the electronic device, instead of targeting the power consumption in the power analysis attacks, which depends on the processed data and the performed cryptographic operations.

In practice, power analysis attacks and electromagnetic attacks are two types of side channel attacks, where the first one targeted the power consumption and the later one target the electromagnetic radiation; both have to use the same statistical techniques to explore some important information to obtain the secret key. So, any technique used to mount power analysis attacks can also be used to mount electromagnetic attacks. There are three types of power analysis attacks:

1. **Simple Power Analysis (SPA)**: is the simplest form of power analysis attack and requires full understanding of the cryptographic algorithm. This type of attack comprises visual examination of the power traces for large scale differences from the cryptographic device, running operations over time. At the algorithm level, cryptographic devices consume different amounts of power based on the operation being processed. From [43] it is possible in some applications to figure out which instructions are executing or perhaps which processed data bits are being changed. If those data bits are part of the secret key, that part of the implementation is more vulnerable to attack. This type of attack might suffer from the amount of noise.
2. **Differential Power Analysis (DPA)**: is more complicated than the SPA. However, DPA is a much more robust type of attack because of the statistical analysis techniques it uses to perform the attack. There are two well-known thoroughly investigated power models currently in use (Hamming weight and the Hamming distance models). The basic concept of DPA exploits the hardware power consumption variations which represent the correlation between the power

consumption measurements and the manipulated data while performing operations using secret keys. DPA attacks use signal processing and error correction properties to overcome the noise problem that cannot be analyzed with SPA. Moreover, DPA uses the statistical analysis techniques which involve the evaluation of the differences between the means of power traces from multiple cryptographic operations to predict the secret key; for more details refer to [6] [43] [44] [45].

3. **Correlation Power Analysis (CPA)**: is a development of DPA applies the key hypothesis which correlates the results (power prediction) of the power consumption model to the actual measured power consumption. Finally, the correct key hypothesis is evaluated by the highest peak of the correlation plot.

1.7 Countermeasures against Side Channel Attacks

Since the first introduction of side channel attacks, finding effective countermeasures to thwart or mitigate them has received enormous attention from the research community. The main idea here is to block or mitigate the information leakage. Since each type of attack depends on different leakage of information (power consumption, electromagnetic radiation, timing variations and cache profile information), different techniques are required to mitigate them.

Power analysis and electromagnetic attacks countermeasures aim to break the correlation between any sensitive intermediate key-dependent value and the leakage power consumption from processing that value, during the execution of the cryptographic algorithm. Hiding and masking are two well-investigated solutions at the algorithmic level that require applying modification to the cryptographic algorithm to reduce that dangerous leakage. On the other hand, based on the required security level, countermeasures can be applied on protocol and hardware levels [46]. In this thesis we focus on the algorithm level.

- Hiding: hiding techniques make the leakage (power consumption or electromagnetic radiation) constant or randomized to eliminate the dependency by several ways including random delay [47], dummy operations and shuffling [48].
- Masking: masking techniques make the leakage dependent on some random values. Masking is the common and widely employed countermeasure to protect block cipher implementations against SCAs especially at the software level. In masking, for every execution of the algorithm, the input data and the secret key (sensitive intermediate variables) are obscured with fresh and randomly selected bits. Therefore, all the computations at the algorithm level are masked until the end of the last round to break the correlation between the secret key and the actual power consumption. The final results are unmasked to retrieve the correct results.

Timing attacks countermeasures aim to eliminate the data-dependent timing variation of executing a cryptographic algorithm with different inputs. In other words, these countermeasures manipulate the cryptographic algorithm implementation to change the run time behavior to be independent of the secret information. Many countermeasures to mitigate timing attacks are investigated, especially for AES, including:

- Constant Time Implementation: having an implementation that consumes the same amount of time to process all inputs eliminates the time variation of processing those inputs.
- Randomize the Execution Time: by adding irregular dummy operations, even re-execute the same input, may consume different amounts of time.

Cache attacks countermeasures aim to remove the relationship between the processed data, during the execution of the cryptographic algorithm, and the effect of memory access on the cache. Many countermeasures to mitigate cache attacks are investigated, especially for AES, including:

- Avoiding Memory Accesses: having a lookup table free implementation (no lookup tables are performed). It is effective, but removing the use of any lookup tables will lead us to replace them with the original logical operations to perform the computations, which incurs a performance overhead.
- Alternative Lookup Tables: using very small lookup tables, this will decrease the probability that a specific memory block will not be accessed during the encryption. This technique will reduce but not eliminate this type of attack, thus it needs to be combined with additional countermeasures.
- Data-Oblivious Memory Access Pattern: instead of avoiding the use of lookup tables one can incorporate a Data-Oblivious Memory Access Pattern technique. The idea is that the memory access pattern must be completely oblivious to the data processed through the encryption process. To perform that, one reads all entries of the related lookup table, in fixed order, whenever any single one is

needed. This technique is effective but very slow, thus secure speedup techniques should be applied to solve the performance problem.

CHAPTER 2

An Efficient Leakage Free Countermeasure of AES using SIMD

AES [49] has become a widely and extensively used encryption primitive in a large variety of applications, from high-end servers such as dedicated cryptographic servers to mobile consumer products. The most successfully mounted attacks on AES are side-channel attacks, which utilize the leakage observations resulting from processing the intermediate variable during the execution of the cryptographic algorithms, such as power consumption, electromagnetic radiation, timing variations and cache profile information (hit and miss rates or access tracer) to retrieve the secret keys. Great efforts have been made to create countermeasures against side-channel attacks. Masking is a widely used countermeasure to defeat differential power analysis (DPA) and differential electromagnetic analysis (DEMA) attacks. Having a constant-time implementation is a sound countermeasure to mitigate timing attacks. One way to mitigate cache attacks is by having an implementation that does not use lookup tables. The current solutions are secure against one type of attack but might be vulnerable to others, cause a serious impact on the performance or memory requirements, which make them impractical for real life applications. Keep in mind, applying the higher order masking require us to manipulate the mask values by adding the masking corrections, where all the computations will be duplicated based on the order we used. For example, if we work on the first order (one mask value involved), all the computations will be duplicated, one for the masked value and the other for mask itself. If we work on the second order (two mask values involved), all the computations will be carried out three times, one for the masked value and one for each mask. Moreover, applying provably secure higher order masking schemes also has additional negative impacts on the performance such as securing the non-linear operation. Furthermore, removing the lookup tables to mitigate cache attacks will lead us to replace them with their original logical operations to compute the needed results. For example, if a lookup table is used to compute the multiplication operation over Galois Field GF(2^8), we have to replace it with the original logical operation as in Algorithm 1 Gmult.

Algorithm 1 Gmult multiplication over GF(2^8)

```
Input: $a$ , $b$
Output: $p = GF_8(a \times b)$

1.    $p \leftarrow 0x00$
2.    For $i = 0$ to $7$ do
3.        $if\ (b\ \&\ 1)$              // Check if  LSB of b=1
4.            $p \leftarrow (p \oplus a)$       // p ^= a
5.        $carry \leftarrow (a\&0x80)$ //check  if MSB = 1
6.        $a \leftarrow (a \ll 1)$          // shift a left by 1
7.        $if\ (carry\ \&\ 0x80)$    // if MSB = 1
8.            $a \leftarrow (a \oplus 0x1B)$ // Modular reduction
9.        $b \leftarrow (b \gg 1)$          // Shift right b by 1
```

Finally, achieving a constant-time implementation to defeat timing attacks imposes adding some dummy operations, to make sure all the input data will complete all the computations simultaneously. All of that will imply a negative impact on the performance of our countermeasure. Remarkably, we face three challenges, the security of AES against multiple SCAs at the same time, the performance impact and the memory overhead.

2.1 Higher-order Masking of AES

The idea behind the higher-order masking scheme is to choose *d*, where *d* is the order of the masking scheme, uniformly distributed and independent from any sensitive data random values. Then, the aim is to break the correlation between any sensitive intermediate value *x* and the leakage information from processing that value, and throw the computation time of the algorithm being used, by splitting *x* into *d + 1* tuples or shares, satisfying a group of operations $\perp$ as a relation between them.

$$x = x_0 \perp x_1 \perp \cdots \perp x_d \qquad (1)$$

In (1) $x_1, \cdots, x_d$ shares are commonly called masks and x_0 is the sensitive masked variable. In our work the exclusive-or $\oplus$ is used for $\perp$. Protecting a block cipher such as AES using higher-order masking requires modifying the design of the algorithm and manipulating the masked values, to ensure that applying the same group of operations on the final shares will extract the correct expected cipher text without masking. Taking into account the security property and reducing the penalty of using higher-order masking will guarantee a practical implementation of a countermeasure against any SCA of order less than or equal to *d*.

From the previous chapter, it is clear that AES comprises linear (e.g., ShiftRows, MixColumns, and AddRoundKey) and nonlinear (e.g., SubBytes, also known as S-box in the case of AES) transformations. Linear transformations are easy to mask considering:

$$f(x) = f(x_0) \oplus \cdots \oplus f(x_d) \qquad (2)$$

Thus, the linear transformations can be securely implemented independently on each share in a straightforward way, where $f(\bullet)$ indicates any linear operation. In contrast to the ease of masking the linear transformations, masking the nonlinear S-box transformation in any block cipher is the hardest and most expensive part.

Algorithm 2 SecMult d^{th} order secure mult in F_2^8 **[13]**

Input: shares a_i where $\oplus_i\ a_i = a$, b_i where $\oplus_i\ b_i = b$

Output: shares c_i where $\oplus_i c_i = ab$

```
1.    For $i = 0$ to $d$ do
2.        For $j = i+1$ to $d$ do
3.            $r_{i,j} \leftarrow rand(128)$
4.            $r_{j,i} \leftarrow (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
5.    For $i = 0$ to $d$ do
6.        $c_i \leftarrow a_i b_i$
7.        For $j = 0$ to $d, j \neq i$ do
8.            $c_i \leftarrow c_i \oplus r_{i,j}$
```

2.2 Provably Secure Higher-order Masking of AES S-Box (nonlinear)

Ishai, Sahai and Wagner's ISW scheme [50] masks Boolean AND gates combined with a NOT gate in any cryptographic operational circuit. The scheme requires each input bit to the AND gate to be split into *2d+1* shares, where *d* is the order of the masking scheme.

Rivian and Prouff [13] generalized the ISW technique by masking the AND gate that represents the multiplication in GF(2), to design a provably secure d^{th}d$^{\text{th}}$ order multiplication in GF(2^8) (see Algorithm 2). In addition, they reduced the number of required shares from *2d+2* to *d+1*. Furthermore, they proposed the first secure d^{th} order S-box to involve the inversion over GF(2^8) based on the exponentiation operation. As the inverse x^{-1} of a polynomial-based representation value $x \in$ GF(2^8) is hard to compute using division over GF(2^8), they found an addition chain that needs four secure multiplications of SecMult as in Algorithm 2 and seven secure squares over GF(2^8) to calculate $x^{-1} = x^{254}$. Finally, they used a lookup tables (LUTs) approach to implement multiplication and squares over GF(2^8).

2.3 Our Contributions

In the work, we definitely aim for producing a countermeasure of AES against multiple side channel attacks at the same time with a reasonable and practical speed. We focus on techniques to produce an efficient leakage-free countermeasure of AES against power analysis, electromagnetic emissions, timing and cache attacks as well as techniques to improve the performance and solve the memory overhead issue. This countermeasure features a secure higher-order masking scheme (to defeat power and electromagnetic attacks), the elimination of lookup tables (to mitigate cache attacks and remove memory overhead), and a constant-time implementation (to defend against timing attacks). However, combining these countermeasures imposes a negative impact on the performance. Therefore, we apply techniques to solve the performance problem at the algorithm and data levels.

2.4 Polynomial Over GF(2^8)

In this section, we work on the original domain where all the calculations are carried out on Galois field GF(2^8). Based on our analysis, the higher order masking scheme of AES spends the most time computing the SecMult operation that calls the Galois field multiplication in the field GF(2^8), which is used in the masked S-box operation when we compute the inverse operation in the SubByte transformation and when we multiply in the field GF(2^8) by (02) and (03) in the MixColumn transformation. To solve the performance problem without losing the SCA immunity, we applied improvements in two levels:

1- **Algorithmic level**: Combine some fast, short and provably high order secure techniques against three types of SCA.

2- **Data level**: Take the advantage of SIMD technology. Here we process 16 bytes at the same time, rather than one by one, and we will pay some penalty to make that work using the existing SSSE3 instructions.

Higher-order masking of AES involves two types of operation, as mentioned earlier: linear and nonlinear. The linear operations are easy to mask. The nonlinear operation (e.g., SecMult over GF(2^8)) is the most expensive operation to evaluate in the whole cipher. Thus, increasing the speed of the higher-order masking scheme relies on decreasing the time take to evaluate the S-box transformation.

2.4.1 Algorithmic Level

A. Secure S-box (SecSbox)

We need to reduce the instances of using the secure multiplication (SecMult) function as much as possible. We chose the d^{th} order SecSbox operation based on the exponentiation operation that was proposed by Rivain and Prouff in [13]. They found an efficient addition-chain that minimizes the number of SecMult and uses a secure square (SecSquare) operation as follows:

$$x \rightarrow x^{2} \rightarrow x^{3} \rightarrow x^{12} \rightarrow x^{15} \rightarrow x^{240} \rightarrow x^{252} \rightarrow x^{254} \qquad (3)$$

In the above chain, we have seven squares and four multiplication operations [13]. The SecSquare is fast because it is linear and needs a less number of multiplications in Galois field than the normal SecMult.

B. MixColumn:

In the MixColumn transformation we need to multiply by (02) and by (03) in F28. However, keep in mind that the identity in (4) can help us to reduce the number of SecMult function usage. Therefore, instead of multiplying x by (02) and then by (03) we can multiply x by (02) then XOR- the addition operation in F_2^8- the result with x again to get the x multiply by (03) as follows:

$$(03)\cdot x = \{(02)\cdot x\} \oplus x \qquad (4)$$

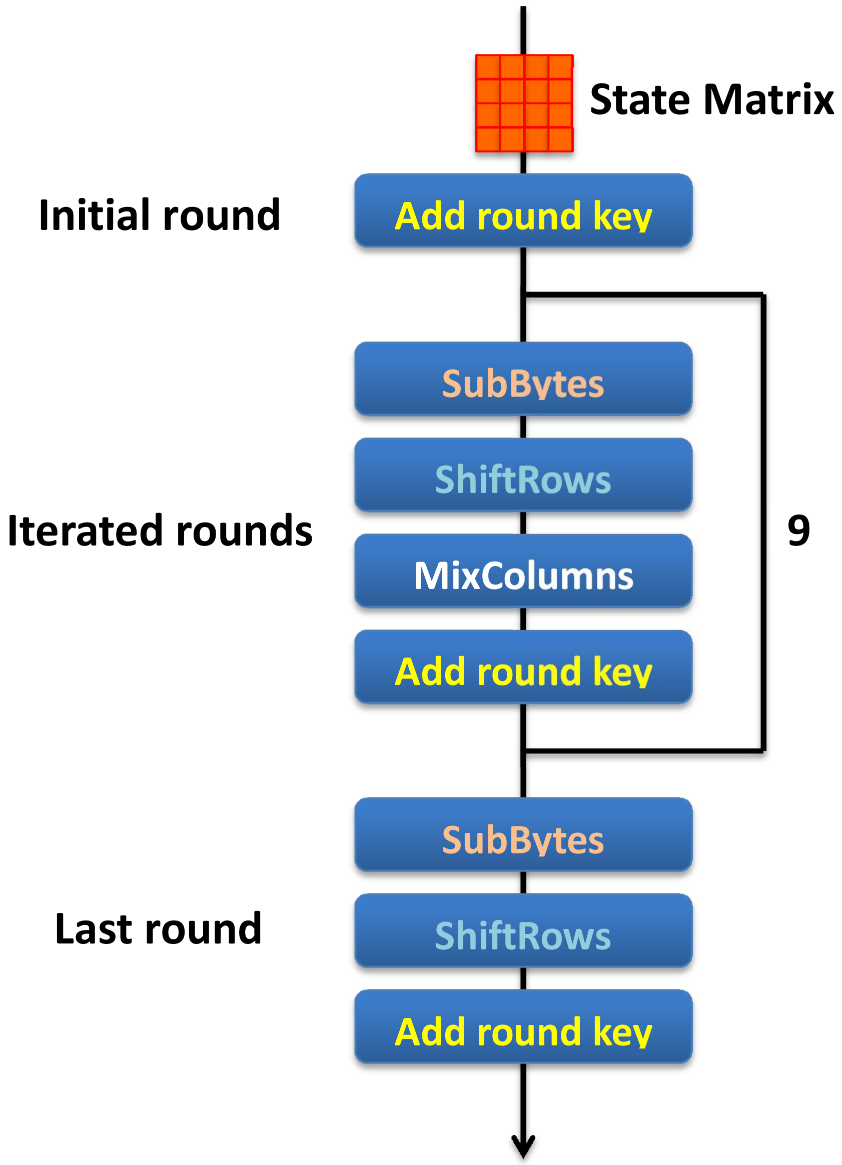


Figure 6: AES Operations

2.4.2 Parallel Computation using SIMD technology

In Intel machines, SSSE3 instruction set comprises a special type of instructions that permit programmers to take advantage of data-level parallelism, using a single instruction to manipulate multiple data simultaneously. The new independent XMM register set is used to execute such instructions; each XMM register is a 128-bit length, which allows programmers to process sixteen bytes (eight bits each) at the same time. Using SIMD technology ensures the number of CPU clock cycles (running time) is reduced and the power consumption of any program is appropriate for data level parallelism. In this proposed technique, we will employ the SIMD technology in all the functions to take advantage of data level parallelism as shown in Figure 6, specifically SecMult, which is the non-linear function. One of the challenges is how to manipulate each byte without affecting the other byte next to it in the same XMM register. For example, when we do the shift left in the GF(2^8) multiplication function, we do not want any shifted byte to affect the byte next to it.

Fortunately, the state matrix consists of 16 bytes (8 bit each); in this case, we can fit all of the state matrix bytes in one XMM register. Then depending on the order, we will have mask values in front of each state matrix element. For example, if $d = 1$ we will have one mask for each element which means 16 mask values that can be fit in another XMM register to perform the mask correction in parallel whenever we need.

In our technique, to perform an AES d^{th} order masking computation, we will assume passing the $d+1$ shares of the masked secret key to the algorithm as an input (to mitigate first order attacks). It is also worth, at this point, to divide the operations in the

AES masking scheme into linear and nonlinear operations. The linear operations include AddRoundKey, ShiftRows, MixColumn, SecSquare, and AffineTransformation. Linear operations are easier to mask and can be performed on each share separately. The only nonlinear operation, on the other hand, is the SecMult function, which is the most expensive function and can be masked using Algorithm 2.

A. Parallelize Galois field (2^8) Multiplication using SIMD

The multiplication function (Gmult) performs multiplication over GF(2^8) of two 128 bit XMM register each contains 16 bytes (8 bit each). This function multiplies each byte in the first XMM register by the byte in the same position of the second XMM register. To perform this multiplication, we recall Shift-and-Add method that was proposed in [51] to calculate the multiplication in polynomial over GF(2^8) and then manipulate it to be applicable to work on XMM registers.

Here *a* (the Multiplicand) and *b* (the Multiplier) are the two inputs to be multiplied. The order does not matter since we will get the same result. To perform the binary multiplication, we need to scan and test each bit of the multiplier right to left. Then, depending on whether the bit is 1 or 0, we will add the multiplicand to the product (*p*) or not. After each test, we have to shift the multiplicand left by one and shift the multiplier right by 1. The result will be stored in *p*. The function can be described with pseudo codes as in Algorithm 1.

In order to perform the multiplication function on XMM registers, we have to make sure that none of the bytes will affect the one next to it, especially when we do the shift left or right operations. Therefore, we used "paddb" to perform the shift left operation for the 16 bytes in parallel (Line 6). Also, we shift left the mask of *b* instead of shift *b* right (Line 9) because shift right in SIMD is expensive. Another concern, is how to check the LSB (Least significant bit) to perform (*p* ^= *a*) (Line 4) and MSB (Most significant bit) of each byte to detect if the carry is about to be generated and then apply the reduction by XOR with (0x1B) (Line 8), thus we do not expand the number of bits more than 8 bits for each byte. This issue has been resolved by using the "pcmpeqb" instruction, which performs the parallel comparison (Lines 3 and 7).

B. Parallelize the Remaining Linear Operations using SIMD

AddRoundKey is implemented using one “pxor” instruction that performs sixteen parallel XOR operations.

ShiftRows is a function where we do not shift the first row but we shift the second, third and fourth by 1, 2, 3 bytes, respectively. Thus, it is constructed using one “pshufb” instruction with Mask = (0x03020100, 0x06050407, 0x09080b0a, 0x0c0f0e0d).

MixColumns In this function we multiply (over Galois field F_2^8) each column of four bytes with a constant matrix as in (5).

$$\begin{bmatrix} S_{0,i} \\ S_{1,i} \\ S_{2,i} \\ S_{3,i} \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 2 \\ 3 & 1 & 1 & 3 \end{bmatrix} \rightarrow \begin{bmatrix} S'_{0,i} \\ S'_{1,i} \\ S'_{2,i} \\ S'_{3,i} \end{bmatrix} \qquad (5)$$

Multiplication by (02) is a special case of Algorithm 1, it is simply a shift left and a conditional XOR that can be implemented as a separate function Gmult_by2 because it will take smaller time than the normal Gmult (Algorithm 1) function. The later one will do the shift 8 times. However, Gmult_by2 needs only one shift and then check if we have to perform the reduction (by XOR with 0x1B) or not. Therefore, one “paddb” instruction is used to perform the parallel shift left operation and one “pxor” instruction is used for the modular reduction if needed, based on the decision of the “pcmpeqb” instruction. Multiplication by (03) is achieved by XORing (using one “pxor” instruction) the multiple by (02) with the original value.

Let us consider a single column of the state and apply the MixColumn transformation on it. We will get:

$$\begin{aligned}
S'_{0,i} &= (2\ .S_{0,i}) \oplus (3\ .S_{1,i}) \oplus S_{2,i} \oplus S_{3,i} \\
S'_{1,i} &= S_{0,i} \oplus (2\ .S_{1,i}) \oplus (3\ .S_{2,i}) \oplus S_{3,i} \\
S'_{2,i} &= S_{0,i} \oplus S_{1,i} \oplus (2\ .S_{2,i}) \oplus (3\ .S_{3,i}) \\
S'_{3,i} &= (3\ .S_{0,i}) \oplus S_{1,i} \oplus S_{2,i} \oplus (2\ .S_{3,i})
\end{aligned} \qquad (6)$$

Now, let us rewrite the single column of the state as shown in (6) and apply the MixColumn transformation on it taking into our accounts that by applying the above identity (4) we will get:

$$\begin{aligned}
S'_{0,i} &= (2\ .S_{0,i}) \oplus \{(2\ .S_{1,i}) \oplus S_{1,i}\} \oplus S_{2,i} \oplus S_{3,i} \\
S'_{1,i} &= S_{0,i} \oplus (2\ .S_{1,i}) \oplus \{(2\ .S_{2,i}) \oplus S_{2,i}\} \oplus S_{3,i} \\
S'_{2,i} &= S_{0,i} \oplus S_{1,i} \oplus (2\ .S_{2,i}) \oplus \{(2\ .S_{3,i}) \oplus S_{3,i}\} \\
S'_{3,i} &= \{(2\ .S_{0,i}) \oplus S_{0,i}\} \oplus S_{1,i} \oplus S_{2,i} \oplus (2\ .S_{3,i})
\end{aligned} \qquad (7)$$

Accordingly, we will form the final equations to implement the MixColumn the 16 bytes of the state in parallel as follows:

$$
\begin{aligned}
y_0 &= S_{1,i} \oplus S_{0,i} \oplus S_{0,i} \oplus S_{0,i} \\
y_1 &= S_{2,i} \oplus S_{2,i} \oplus S_{1,i} \oplus S_{1,i} \\
y_2 &= S_{3,i} \oplus S_{3,i} \oplus S_{3,i} \oplus S_{2,i} \\
y_3 &= 2\left[S_{0,i} \oplus S_{1,i} \oplus S_{2,i} \oplus S_{0,i}\right] \\
y_4 &= 2\left[S_{1,i} \oplus S_{2,i} \oplus S_{3,i} \oplus S_{3,i}\right] \\
S'_{Mix} &= y_0 \oplus y_1 \oplus y_2 \oplus y_3 \oplus y_4
\end{aligned}
\qquad (8)
$$

Thus, we need 5 "pshufd" instructions to do the reordering and then four "pxor" to get the final result.

AffineTransformation is performed after the multiplicative inverse operation to complete the S-box transformation. It is the sum (XOR operation) of four rotations of the byte as a vector. Suppose that we got the result of one byte from the Multiplicative Inverse in a variable $x = [x_0, x_1, \ldots, x_7]$ where $x_0, \ldots, x_7$ are bits, to implement the affine transformation we need to follow Algorithm 3.

Algorithm 3 Affine_Trans: affine transformation

```
Input: $x$
Output: $c$ = affine transformation of ($x$)

1.    $y = x$.
2.    For $i = 0$ to $3$ do
3.        $x \leftarrow x \oplus \{rotate\_left(y)\ by\ one\ bit\}$
4.    $c \leftarrow x \oplus 0x63$
```

Thus, we do not have to check if it is even or not to avoid conditional branches in order to ensure a constant operation flow [13]. To securely apply the masking scheme on this linear function, each share can be transformed independently except XOR with 0x63 in (line 4), that need to be XORed with one share only.

2.4.3 Masking the Parallelize operations

From the previous section, we succeed in applying the SIMD technology on all the AES operations to archive data level parallelism. Accordingly, now our solution can perform sixteen parallel computations simultaneously instead of one by one. Now the next step is to mask those computations to secure them. As mentioned in Section 2.1, AES comprises linear and non-linear operations. The linear operations are easy to mask considering (2), where applying the operation on each share separately is secure in the higher order masking scheme. For example, masking the field squaring (SecSquare), where we securely compute the square of the state shares, is accomplished by square every share separately by multiply the share by itself as shown in Algorithm 4. Thus, it requires ($d+1$) calls to Gmult function in Algorithm 1.

Algorithm 4 SecSquare - d^th-order secure square over F_2^8

```
Input: shares $a_i$ satisfying $\oplus_i a_i = a$
Output: shares $c_i$ satisfying $\oplus_i c_i = a^2$

1.    For $i = 0$ to $d$ do
2.        $c_i \leftarrow (a_i)^2$
```

However, masking the non-linear operation is the critical and most expensive part of the whole algorithm. For example, masking the field multiplication (SecMult), where our technique adapts the same SecMult function that was proposed in [13], which uses Ishai-Sahai-Wagner (ISW) scheme. Except that we do the multiplication using our Gmult

(Algorithm 1) function instead of using the look-up tables (LUTs) approach. Thus, Algorithm 2 needs *$(d+1)^2$* calls to Gmult (Algorithm 1) function, *$2d(d+1)$* XORs and *$d(d+1)/2$* random 128-bit values to be generated.

From the two examples above, it is obvious that the linear operations are easier to mask compared with the non-linear operation.

2.5 Security Analysis

The overall security of our implementation can be easily proved using the proofs in [13]. Moreover, we believe that we solved the differential power and electromagnetic analysis (by applying the high order masking scheme), the timing attack (by achieving a constant-time implementation as a result of using SIMD technology), and the cache-attack (by avoiding the use of any look-up tables).

2.6 Implementation Results

To achieve the best optimization, we implemented our scheme (higher-order masking of AES) by leveraging SSSE3 instructions with C-language in Linux machine with SSSE3 support. Moreover, using inline assembly code gave us the best performance. We assumed the availability of (an inaccessible to attackers) a pool of independent and uniformly distributed random masks. We compared the implementations of our scheme with two different implementations [13] [52] interims of the first, second and third order. In [13], they implemented their scheme and used log/ alog table to improve the efficiency of the multiplication over F_2^8 which is the most costly function. In [52], they did the exponentiation operation over the composite field F_2^4 to get better performance and they used six lookup tables to speed up the squaring, two squaring, squaring-scalar multiplication, multiplication, isomorphism and inverse isomorphism operations. In our implementation, we masked the entire rounds and we do not use any lookup tables. However, we gain a speedup of 9.5, 9.3 and 7.6 times faster than the existing countermeasure in [13] for first, second and third order masking. Also, we gain a speedup of 1.1, 6.8 and 5.6 times faster than the existing countermeasure in [52] for first, second and third order masking as listed in Table 1.

Table 1. Comparing Encryption Execution Time and Size of Lookup Tables with Other Schemes

Scheme	Cycles ($\times 10^3$cc)	Table size(Bytes)	Times
	Encryption	**LUT**	**Speedup**
Original AES (Straightforward AES)			
[52]	9.0	255	
First Order Masking			
[13]	129	3153	9.56
[52]	14.9	256	1.1
Ours	13.5	0	
Second Order Masking			
[13]	271	3845	9.34
[52]	199.3	816	6.87
Ours	29	0	
Third Order Masking			
[13]	470	4648	7.62
[52]	346.8	816	5.62
Ours	61.7	0	

CHAPTER 3

Efficient SubByte Transformation (S-box)

Again higher-order masking of AES involves two types of operation, as mentioned earlier: linear and nonlinear. The linear operations are easy to mask. The nonlinear operation (e.g., SecMult) is the most expensive operation to evaluate in the whole cipher. Furthermore, SecMult is one of many functions in which evaluating the AES S-box has to perform. Thus, increasing the speed of the higher-order masking scheme relies on decreasing the time take to evaluate the SubByte (known also as S-box) transformation. Thus, this chapter focuses on incorporate some techniques to accelerate the bottleneck transformation of the whole AES the SubByte transformation, which involves two major operations; the inversion and the affine transformation, aiming to achieve better performance when applying the Higher-order masking scheme. Optimizing the S-box using composite field has been proposed in [53].

3.1 Our Contributions

Our objective is to increase the performance of the bottleneck transformation (SubByte transformation) with the same level of security by apply enhancement techniques in three different levels:

1. **Algorithm level**: Combine some fast and provably high order secure algorithms. To achieve that we will transform all the S-box computations from GF(2^8) to map

x to GF(2^4), where the inverse is evaluated using the power function $x^1 \rightarrow x^{254}$ over GF(2^8) as in (3), while in GF(2^4) using the power function $x^1 \rightarrow x^{14}$ as in (11) map *x* to GF(2^4). More over each element in GF(2^4) is represented by 4-bits compared with 8-bits in GF(2^8), which means the Galois field multiplication will coast us have of the time in GF(2^4) that in GF(2^8) as a result of reducing the number of iterations from 8 as in Algorithm 1 to 4 as in Algorithm 5.

2. **Byte level**: Take advantage of the parallel computing by using Single Instruction Multiple Data (SIMD) technology (SSSE3 instructions).
3. **Bit level**: Driving the formulas for each bit if (applicable).

$$\delta : GF(2^8) \rightarrow GF((2^4)^2)$$

$$\delta = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9)$$

3.2 Inversion Operation Over Composite Field GF(2^4)

In this section, we work on the extension field (composite field) domain where all the calculations are carried out over GF(2^4). The idea is to transform each element in the finite field GF(2^8) to the finite field GF((2^4)^2) because the finite field GF(2^8) is isomorphic to the finite field GF((2^4)^2). In other words, there exists only one element in GF((2^4)^2) for each element in GF(2^8). This transformation from any element $a \in$ GF(2^8)

to a two-term polynomial $a_h x + a_l$ where a_h and $a_l \in$ GF(2^4) or vice versa is achieved by multiplying with the isomorphism functions in (9) and (10).

$$\delta^{-1} : GF((2^4)^2) \rightarrow GF(2^8)$$

$$\delta^{-1} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad (10)$$

Figure 7 shows the inverse operation that is the first step in the AES S-box operations over the composite field. The affine transformation is then applied to complete the S-box computation. For more details refer to [53].

To perform the inverse operation over the composite field shown in Figure 7 we have to execute the following steps:

Step1. Map x to GF(2^4)MapxtoGF(2^4) by multiplying x by δ to perform a_h and $a_l \in$ GF(2^4).

Step2. Produce $d = a_h^2 \lambda + a_l(a_h + a_l) \in$ GF(2^4).

Step3. Compute the inverse $d' = d^{-1} \in$ GF(2^4) using the exponentiation operation.

Step4. Extract $a_h' = d' a_h$, $a_l' = d' a_l \in$ GF(2^4).

Step5. Map^{-1} (a_h', a_l') to GF(2^8) by multiplying (a_h', a_l') by δ^{-1} to perform $x^{-1} \in$ GF(2^8).

We can securely mask all the linear functions from step 1 to step 5 (mapping δ, δ^{-1}, squaring, adding ⊕ and scalar multiplication by λ) by applying the operation on each share independently, contrary to the linear operation (multiplication over GF(2^4)MapxtoGF(2^4)) which must follow Algorithm 2. In step 3, the inversion over GF(2^4)MapxtoGF(2^4) is evaluated using the addition chain in (11) that was proposed in [52]:

$$x^1 \xrightarrow{S} x^2 \xrightarrow{M} x^3 \xrightarrow{2S} x^{12} \xrightarrow{M} x^{14} \qquad (11)$$

This chain contains a series of linear operations (single square (S) and double square (2S)) and two nonlinear operations (multiplication (M)). All squares and multiplications are over GF(2^4)MapxtoGF(2^4). In [52] the authors implemented the masking using six precomputed lookup tables to enhance the computation speed of squaring, two squaring, squaring-scalar multiplication, multiplication, isomorphism and inverse isomorphism functions. That is exactly what we would like to avoid, in order mitigating cache attacks (our technique does not use any lookup tables).

3.3 Parallelize Galois field (2^4) Multiplication using SIMD

As stated earlier, the AES block comprises of 16 bytes. We desire to reach data-level parallelism by design the Gmult function, which can perform sixteen multiplications over GF(2^4) simultaneously. Galois field multiplication over GF(2^4) involves multiplication of two polynomials $a(x)$ and $b(x) \in$ GF(2^4) and modular reduction of the product with $m(x)$ =

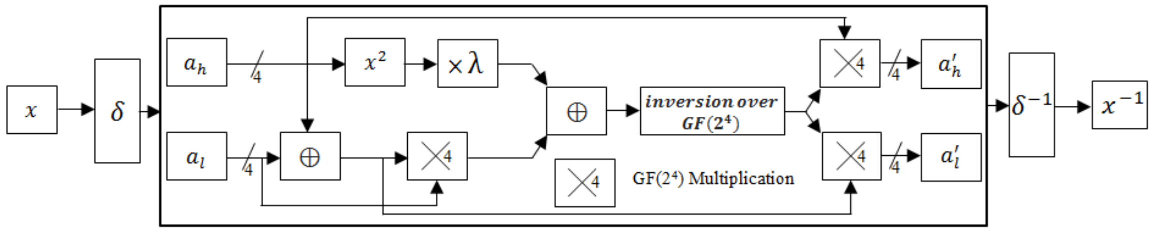


Figure 7: Inverse operation over composite field [41]

$x4 + x + 1 \in \mathrm{GF}(2^4)$. We designed the Gmult function, which requires two parameters of type *_m128i* that can be held in two 128 bit XMM registers each containing 16 bytes (8 bits each), and returns the result of the multiplication in a 16 bytes XMM register following Algorithm 5.

Algorithm 5 Gmult multiplication over $\mathrm{GF}(2^4)$

Input: $a, b \in \mathrm{GF}(2^4)$
Output: $p = \mathrm{GF}_4(a \times b)$

```
1.    $p \leftarrow 0x00$
2.    **For** $i = 0$ **to** $3$ **do**
3.        $if\ (b \,\&\, 1)$
4.            $p \leftarrow (p \oplus a)$
5.        $carry \leftarrow (a \& 0x80)$
6.        $a \leftarrow (a \ll 1)$
7.        $if\ (carry \,\&\, 0x80)$
8.            $a \leftarrow (a \oplus 0x13)$
9.        $b \leftarrow (b \gg 1)$
```

Algorithm 5 explains how to evaluate a multiplication in $\mathrm{GF}(2^4)$. First, reset the product $p = 0$ by XORing p with itself using the “pxor” instruction, which performs sixteen parallel XOR operations (Line 1). Then, from right to left examine the least significant bit (LSB) in b with the help of the parallel compare “pcmpeqb” instruction (Line 3). Next, for each non-zero bit, add a to the product (Line 4). Detect if the carry is about to be generated (Line 5), as a result of shifting a left, through the parallel AND bytes “pand” instruction; this step will detect any byte in which the most significant bit (MSB) equals one. Next, a is doubled. Shift a left by one bit via the parallel add bytes "paddb" instruction that performs the parallel shift left operation (Line 6). The "pcmpeqb" instruction is used once more to produce the masks to perform the conditional operations (Line 7). The results of the "pcmpeqb" instruction (Lines 3 and 7), either 0xFF or ox00, are used as masks to decide whether or not to perform the operations in line 4 and 8. A modular reduction is performed (Line 8) by XORing *a* with *m*(*x*). Finally, *b* is shifted right by one bit, which is costly to implement using SSSE3 instructions; instead, we shift the mask of *b* to the left using "paddb" instruction (Line 9).

3.4 Parallel and Masked Inversion over Composite Field

As mentioned above, the inversion over a composite field comprises two different types of operations, linear and nonlinear. All the operations should satisfy the provably secure higher-order security conditions. We design our d^{th} order masked inversion over a composite field that fulfils these requirements. First, we divide the inversion operations in Figure 7 into seven steps, as shown in Figure 8.

In Figure 8, we classify the linear (steps 1, 2a, 2b, 4 and 7) and nonlinear (steps 3, 5 and 6) operations. It is straightforward to secure the linear operations by applying the operation on each tuple separately because they satisfy (2). Iimplementing step 1, which maps any polynomial element from GF(2^8) to the two-term polynomial $a_h x + a_l$ where a_h

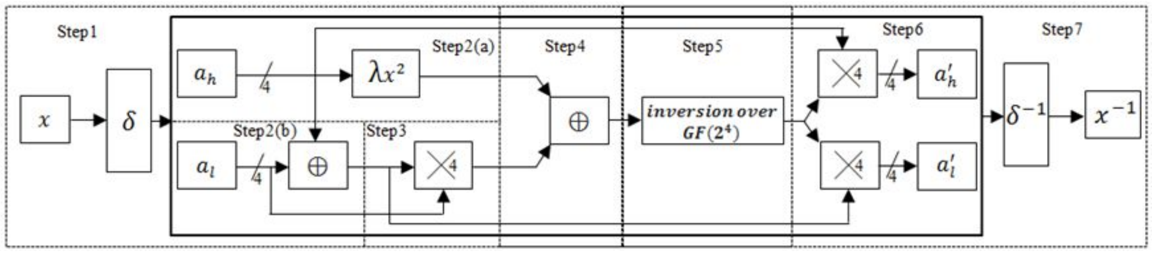


Figure 8: Parallel and masked inverse operation over composite field

and $a_l \in$ GF(2^4) and vice versa (step 7) mapping back in a secure masked way, is accomplished by multiplying by δ and δ^{-1} respectively for each share independently.

Masking the XOR operation (i.e., steps 2b and 4) is achieved using the “pxor” instruction for each share separately.

For further enhancements, we combine some of the linear operations in one function to reduce the execution time. For instance step 2(a) in Figure 8, (λx^2) is a combination of two functions, squaring and multiplication by lambda.

Masking the squaring with multiplication by lambda (λx^2), which is also a linear operation, can be performed on each share independently.

For masking the field multiplication over GF(2^4) multiplication is the critical and most expensive part of the whole cipher; the nonlinear operation. We recall the *SecMult4* function that was proposed in [52], which follows Algorithm 2, except that we use our Gmult function (Algorithm 5) instead of using the LUT approach.

Masking the inversion over GF(2^4) (step 5) requires raising each element to the power 14 as described in (11). To design this function, three different operations are used: squaring, double squaring (raising to the power 4) and multiplication. Both the square and double square functions are linear. Therefore, it is straightforward to mask them by performing the function on each share individually.

3.5 Deriving Formulas

In this section, rather than performing the square in the polynomial representation using the Gmult function as in Algorithm 5, which contains four iterations of checking the least significant bit (LSB) to Xor with the result variable, check if the carry is about to be generated by checking the most significant bit (MSB) to apply the modular reduction with *m(x) = x4 + x + 1* $\in$ GF(2^4), shift a to the left and shift b to the right operations, we will utilize a *Karnaugh map* to drive the formulas for each bit. A Karnaugh map is a special mechanism, using Gray code to make adjacent, bit patterns that differ in one bit position, to lower the required extensive calculations by taking advantage of humans' pattern-recognition capability. This mechanism has been adopted to reduce the amount of required calculations to perform the square operation [54]. The idea is to perform all the computations to do the squaring on all possible inputs as in Table 2. Then, applying a Karnaugh map on each column (see Figure 9, Figure 10, Figure 12, and Figure 12) of the final truth table to get the sum-of-products terms (known as *minterms*) as in (12).

$$\begin{aligned} f(A,B,C,D)_{S_A} &= A \\ f(A,B,C,D)_{S_B} &= AC' \oplus A'C \\ f(A,B,C,D)_{S_C} &= B \\ f(A,B,C,D)_{S_D} &= BD' \oplus B'D \end{aligned} \qquad (12)$$

After that, simplify the resulting minterms using Boolean algebra by making use of relationships and theorems such as grouping, multiplication by redundant variables, DeMorgan's Theorem, etc., to get the final results as shown in (13).

$$\begin{aligned} f(A,B,C,D)_{S_A} &= A \\ f(A,B,C,D)_{S_B} &= AC' \oplus A'C = A \oplus C \\ f(A,B,C,D)_{S_C} &= B \\ f(A,B,C,D)_{S_D} &= BD' \oplus B'D = B \oplus D \end{aligned} \qquad (13)$$

Finally, the square operation, which is a special case of multiplication, can be performed in any element over GF(2^4) using (14):

$$q(x) = a(x)^2 \bmod m(x) \quad \text{where } q(x), a(x) \in GF(2^4) \tag{14}$$

$$q_0 = a_0 \oplus a_2, \quad q_1 = a_2$$
$$q_2 = a_1 \oplus a_3, \quad q_3 = a_3$$

Table 2: Squaring Truth Table

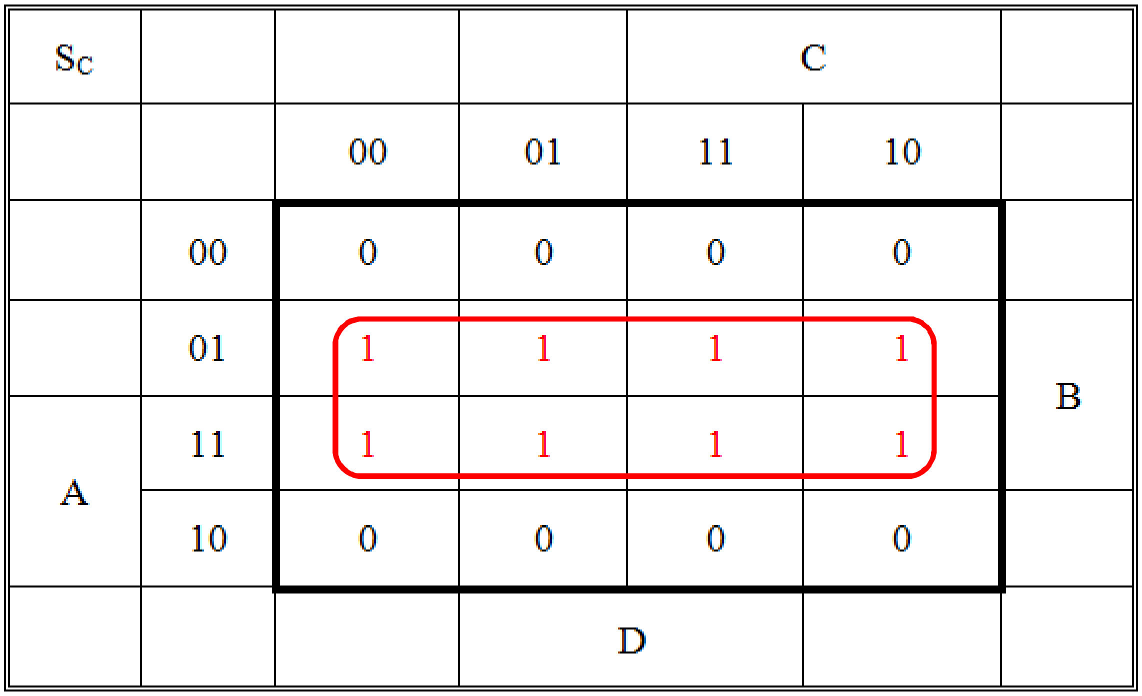
x_d	x				Square(x)			
	A	B	C	D	S_A	S_B	S_C	S_D
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	1	0	0
3	0	0	1	1	0	1	0	1
4	0	1	0	0	0	0	1	1
5	0	1	0	1	0	0	1	0
6	0	1	1	0	0	1	1	1
7	0	1	1	1	0	1	1	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	0	0	0
11	1	0	1	1	1	0	0	1
12	1	1	0	0	1	1	1	1
13	1	1	0	1	1	1	1	0
14	1	1	1	0	1	0	1	1
15	1	1	1	1	1	0	1	0
	a_3	a_2	a_1	a_0	a_3	$a_3 \oplus a_1$	a_2	$a_2 \oplus a_0$

S_A				C		
		00	01	11	10	
	00	0	0	0	0	
	01	0	0	0	0	B
A	11	1	1	1	1	
	10	1	1	1	1	
			D			

Figure 10: Karnaugh map for S_A in Table 2

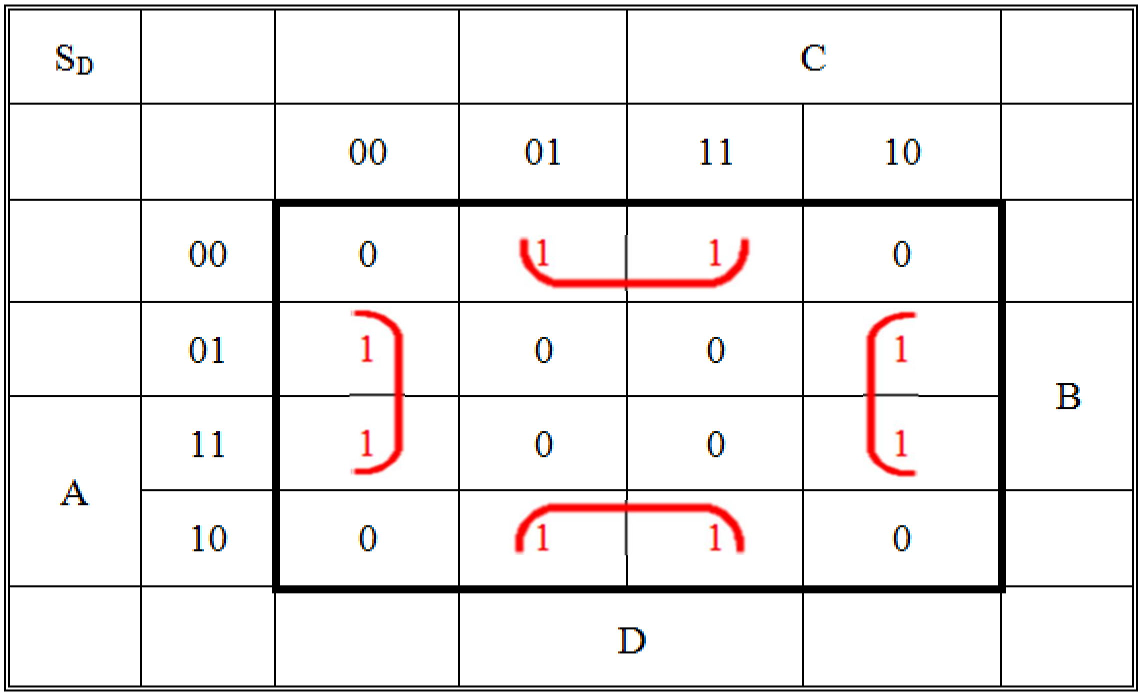
S_B				C		
		00	01	11	10	
	00	0	0	1	1	
	01	0	0	1	1	B
A	11	1	1	0	0	
	10	1	1	0	0	
			D			

Figure 9: Karnaugh map for S_B in Table 2



S_C				C		
		00	01	11	10	
	00	0	0	0	0	
	01	1	1	1	1	B
A	11	1	1	1	1	
	10	0	0	0	0	
			D			

Figure 12: Karnaugh map for S_C in Table 2



S_D				C		
		00	01	11	10	
	00	0	1	1	0	
	01	1	0	0	1	B
A	11	1	0	0	1	
	10	0	1	1	0	
			D			

Figure 12: Karnaugh map for S_D in Table 2

Moreover, using the same technique multiplication by lambda can be performed in any element over GF(2^4) by:

$$\begin{array}{cc}
\multicolumn{2}{c}{q(x) = a(x) \otimes \lambda \bmod\ m(x)} \\
\multicolumn{2}{c}{\text{where } \lambda = 0xe} \\
\hline
q_0 = a_1 \oplus a_2 \oplus a_3, & q_1 = a_0 \oplus a_1 \\
q_2 = a_0 \oplus a_1 \oplus a_2, & q_3 = a_0 \oplus a_1 \oplus a_2 \oplus a_3
\end{array} \qquad (15)$$

Substitute the equations from (14) in (15):

$$q_0 = a_2 \oplus a_1 \oplus \cancel{a_3} \oplus \cancel{a_3}$$

$$q_1 = a_0 \oplus \cancel{a_2} \oplus \cancel{a_2}$$

$$q_2 = a_0 \oplus \cancel{a_2} \oplus \cancel{a_2} \oplus a_1 \oplus a_3$$

$$q_3 = a_0 \oplus \cancel{a_2} \oplus \cancel{a_2} \oplus a_1 \oplus \cancel{a_3} \oplus \cancel{a_3}$$

Finally, squaring and multiplication by lambda can be performed as one function (λx^2) in any element over GF(2^4) by:

$$\begin{array}{cc}
\multicolumn{2}{c}{q(x) = a(x)^2 \otimes \lambda \bmod m(x)} \\
\multicolumn{2}{c}{\text{where } \lambda = 0xe} \\
\hline
q_0 = a_1 \oplus a_2, & q_1 = a_0 \\
q_2 = a_0 \oplus a_1 \oplus a_3, & q_3 = a_0 \oplus a_1
\end{array} \qquad (16)$$

The double square function can be implemented instead of calling the square twice, via combining the square with itself mod *m(x)*. Hence, the double square function is implemented by substituting the equations from (14) in (14) again:

$$\begin{aligned}
q_0 &= a_0 \oplus a_2 \oplus a_1 \oplus a_3 \\
q_1 &= a_1 \oplus a_3 \\
q_2 &= a_2 \oplus a_3 \\
q_3 &= a_3
\end{aligned}$$

Thus, the double square (x^4) can be performed in any element over GF(2^4) by:

$$
\begin{gathered}
q(x) = a(x)^4 \bmod m(x) \\
\text{where } q(x), a(x) \in GF(2^4)
\end{gathered}
\qquad (17)
$$

$$
\begin{array}{ll}
q_0 = a_0 \oplus a_1 \oplus a_2 \oplus a_3, & q_1 = a_1 \oplus a_3 \\
q_2 = a_2 \oplus a_3, & q_3 = a_3
\end{array}
$$

3.6 Affine Transformation

AffineTransformation is performed after the multiplicative inverse operation to complete the S-box transformation by multiplying with the affine transformation (AT) matrix as in (18). Then, XORing with the 0x63 hexadecimal value.

$$
AT = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 1
\end{bmatrix} \qquad (18)
$$

For more enhancements, we merged the isomorphism functions δ^{-1} in (10) with the affine transformation (AT) matrix in (18) to compose a new matrix that performs both operations in the cost of one only (δ^{-1}+AF) as in (19). Therefore, by multiplying the outcome from the multiplicative inverse with this matrix, then XORing the result using the "pxor" instruction with 0x63 with one of the shares only in the case of d^{th} order masking, we complete the S-box transformation.

$$\delta^{-1} + AT = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad (19)$$

3.7 Security Analysis

From [52], the security of our technique can be easily proven. Furthermore, we believe that our countermeasure can prevent power analysis, electromagnetic, timing and cache attacks because we apply the higher-order masking scheme, achieving a constant-time implementation (through using SSSE3 instructions) and avoid the use of any lookup tables.

3.8 Implementation Results

We implement this efficient higher-order masking of AES S-box by incorporating SSSE3 instructions with C-language. Rather than using the intrinsic functions that are easier to deal with, we chose inline assembly code for better performance. We replaced the S-box that was used in CHAPTER 2. After that, we compared our implementation with two different implementations in [13] and [52]. In [13], a higher-order masking scheme was implemented using a log table to improve the efficiency of the multiplication over GF(2^8). In [52], the exponentiation operation was done over the composite field GF(2^4) for better performance and six lookup tables were used to speed up six operations as mentioned above.

In our technique the S-box is computed over the composite field GF(2^4) and no lookup tables are used. However, as shown in Table 3, we gain a speedup of 12.77, 13.48 and 10.11 times faster than the existing countermeasure in [13] for first, second and third order masking, respectively. Speedup is 1.5, 9.9 and 7.5 times faster than the existing countermeasure in [52] for first, second and third order masking.

Table 3. Comparing Encryption Execution Time and Size of Lookup Tables with Other Schemes

Scheme	**Cycles ($\times 10^3$cc)**	**Table size(Bytes)**	**Times**
	Encryption	**LUT**	**Speedup**
Original AES (Straightforward AES)			
[52]	9.0	255	
First Order Masking			
[13]	129	3153	12.77
[52]	14.9	256	1.48
Ours	10.1	0	
Second Order Masking			
[13]	271	3845	13.48
[52]	199.3	816	9.91
Ours	20.1	0	
Third Order Masking			
[13]	470	4648	10.11
[52]	346.8	816	7.46
Ours	46.5	0	

CHAPTER 4

Efficient Multiplicative Inverse

Masking is a widely employed countermeasure to protect block cipher implementations against SCAs especially at the software level. In masking, for every execution of the algorithm, the input data and the secret key (sensitive intermediate variables) are obscured with fresh and randomly selected bits. Therefore, all the computations at the algorithm level are masked until the end of the last round to break the correlation between the secret key and the actual power consumption. As mentioned earlier, the bottleneck transformation of the whole AES, the S-box transformation, which involves two major operations, the inversion and the affine transformation, is the focus of any research to increase the speed of the higher-order masking scheme. As in (3), the efficient addition-chain that minimizes the number of SecMult and uses SecSquare instead carries out seven SecSquare and four SecMult. In the previous two chapters we spent a good amount of effort enhancing the whole cipher including the SecMult. Thus, this chapter focuses on incorporating some techniques to accelerate the SecSquare operation, which is used seven times to evaluate the multiplicative inverse operation that is the heart of the S-box transformation, aiming to achieve better performance when applying the Higher-order masking scheme. Furthermore, this chapter consolidates this new technique to use secure lookup tables that are efficient against cache attacks instead of avoiding the use of lookup tables.

4.1 Our Contributions

Our objective is to increase the performance of the first operation in the bottleneck transformation (S-box transformation), the multiplicative inverse operation, with the same level of security by:

1. Utilize the squaring property of normal basis [55].
2. Solving the basis conversion problem and lowering the required extensive calculations using Karnaugh maps [54] and Boolean algebra.
3. Utilizing SIMD technology, which can manipulate 16 bytes concurrently, to achieve data level parallelism [56].
4. Taking advantage of the "pshufb" instruction, which performs Data-Oblivious Memory Access Pattern technique, to design enhanced countermeasures against cache attacks, instead of avoiding the use of Lookup Tables.

4.2 Finite Field GF(2^8) and Normal Basis

Understanding finite field arithmetic leads to a significant enhancement in implementing cryptographic operations. Conventionally, GF(2^m) arithmetic is classified based on the element basis representation of the finite field. There are two popular bases commonly used in representing finite field elements to implement cryptographic operations; polynomial and normal bases with special mathematical characteristics.

The polynomial basis representation of the finite field GF(2^8) is a basis of form $\{1, x, x^2, \cdots, x^7\}$, where the field element $a_7x^7 + a_6x^6 + \cdots + a_1x + a_0$ is indicated by the coefficient bit string $(a_7a_6 \cdots a_1a_0)$ of length 8, with particular addition and multiplication rules. In fact, polynomial basis representation performs the binary carry-less multiplication efficiently.

The normal basis representation of the finite field GF(2^8) is a basis of form $\{\beta, \beta^{2^1}, \beta^{2^2}, \cdots, \beta^{2^7}\}$ where the transformation of the basis elements is defined by applying the 7th power mapping. Thus, given any element $A \in GF(2^8)$ then applying:

$$A = \sum_{i=0}^{7} a_i \beta^{2^i} \text{ , } where \ a_i \in \{0,1\} \quad (20)$$

The field element A is indicated by the coefficient bit string $(a_7 a_6 \cdots a_1 a_0)$ of length 8, with particular addition and multiplication rules. In fact, the normal basis representation performs the squaring efficiently while the multiplication is more complex than in the polynomial basis representation. For more details refer to [55].

4.3 Incorporate Normal Basis Squaring Feature

In this section, we explain our technique to reduce the cost of the provably secure higher-order masking of AES. As mentioned previously, most of the computation time of the higher order masking of AES is consumed in evaluating the S-box transformation that consists of two operations: the multiplicative inverse in Galois field GF(2^8) followed by the affine transformation. The latter is just a matrix multiplication. Therefore, we will focus on the multiplicative inverse that contains a series of squares and multiplications performed in a specific order based on the addition chain as in (3), that has the least number of multiplication and square operations. This chain has seven SecSquare and four

SecMult. Each SecSquare based on (2), which is a linear operation, contains ($d+1$) calls to the Gmult function Algorithm 1.

Thus, all of that will develop a performance problem, which is our goal to solve in this paper, without losing the security property, by working at three levels:

1- *Algorithm level*: utilizing the efficient squaring property of the normal basis representation [55] of the element where the square is just a cyclic shift left.

2- *Byte level*: utilizing Intel Single Instruction Multiple Data (SIMD) technology to achieve data-level parallelism [56].

3- *Bit level*: Deriving the formulas for each bit using Karnaugh maps and Boolean algebra [54].

In this approach, rather than performing the square in the polynomial representation using the Gmult function as in Algorithm 1, which contains eight iterations of checking the least significant bit (LSB) to Xor with the result variable, check if the carry is about to be generated by checking the most significant bit (MSB) to apply the modular reduction with $m(x) = x^8 + x^4 + x^3 + x + 1$, shift a to the left and shift b to the right operations, we will perform the squaring in the normal basis representation. As mentioned before, the normal basis representation is an alternative representation to the polynomial basis where the square is more efficient, the square in normal basis representation is just a cyclic shift left as shown in Algorithm 6, while the multiplication is more complex. Therefore, we decided to do the multiplication in the polynomial basis and the squaring in the normal basis. In order to do that, one must convert each share from polynomial basis to normal basis. In the normal basis, we perform the square. Next, we transform that result to polynomial basis (which is the original representation) to complete any other computations. Accordingly, efficient basis conversion has become of interest.

4.4 Basis Conversion

In AES all the operations are performed in Galois field GF(2^8). Therefore, the goal is to find the fastest way to compute the representation of that element in the normal basis given its representation in polynomial basis. In order to do that, we chose the root $\beta = x^{32}$to compute the rest of the roots$\left(\beta, \beta^2, \beta^{2^2}, \dots, \beta^{2^7}\right)$. Then, given any element a in polynomial basis representation GF(2^8), the normal basis (NB) representation b of that element can be computed using (20). Subsequently, we derived the table of all elements from 0x00 to 0xFF to calculate all the normal basis elements. Then, we applied a Karnaugh map of 8-variables and Boolean algebra on each bit to find the simplified equation of that element that will map the polynomial representation to the normal representation, which resulted in:

$$\begin{aligned}
b_0 &= a_5 + a_4 + a_2 + a_1 + a_0 \\
b_1 &= a_4 + a_2 + a_1 + a_0 \\
b_2 &= a_4 + a_3 + a_2 + a_0 \\
b_3 &= a_6 + a_4 + a_3 + a_0 \\
b_4 &= a_6 + a_3 + a_1 + a_0 \\
b_5 &= a_7 + a_6 + a_2 + a_1 + a_0 \\
b_6 &= a_4 + a_3 + a_2 + a_1 + a_0 \\
b_7 &= a_6 + a_4 + a_2 + a_1 + a_0
\end{aligned} \qquad (21)$$

After figuring out the needed equations to convert from polynomial basis representation to normal basis representation, we needed to find the fastest way to compute the reverse direction from normal basis representation to polynomial basis representation given its representation in normal basis. Our way to do that is by reordering the table that we got from converting the polynomial to normal based on the normal basis. Then, we applied a Karnaugh map of 8-variables and Boolean algebra on each bit to find the simplified equation of that element that will map the normal representation to the polynomial representation, which resulted in:

$$\begin{gathered}
a_0 = b_7 + b_4 + b_2 \\
a_1 = b_6 + b_2 \\
a_2 = b_7 + b_3 + b_2 + b_1 \\
a_3 = b_6 + b_1 \\
a_4 = b_6 + b_4 + b_3 + b_2 \\
a_5 = b_1 + b_0 \\
a_6 = b_7 + b_1 \\
a_7 = b_7 + b_6 + b_5 + b_4 + b_3 + b_2
\end{gathered} \tag{22}$$

A Karnaugh map is a special mechanism, using Gray code to make adjacent, bit patterns that differ in one bit position, to lower the required extensive calculations by taking advantage of humans' pattern-recognition capability. This mechanism has been adopted to reduce the amount of required calculations to perform the basis conversion [54]. The idea is to perform all the computations to do the transformations on all possible inputs. Then, applying a Karnaugh map on each column of the final truth table to get the sum-of-products terms (known as minterms). After that, simplify the resulting minterms using Boolean algebra by making use of relationships and theorems such as grouping, multiplication by redundant variables, DeMorgan's Theorem, etc., to get the final results as shown in (21) and (22).

4.5 Efficient Normal Basis Squaring

As mentioned before the square operation in normal basis is just a cyclic shift left as shown in Algorithm 6:

Algorithm 6: Normal basis squaring GF(2^8)

```
Input: $a \in$ normal basis GF($2^8$)
Output: $p = (a^2)$

    1. $b \leftarrow a$
    2. $a \leftarrow (a \ll 1)$
    3. $b \leftarrow (b \gg 7)$
    4. $p \leftarrow (a \oplus b)$
```

Designing Algorithm 6 with SIMD technology is achieved with the following assembly code after assuming that XMM 1 holds the converted share in the normal basis representation:

```
"movdqa  xmm1, xmm2;"     // xmm2 <- xmm1
"psrlw   $7     , xmm2;"  // shift right by 7 word wise
"pand    xmm8, xmm2;"     // and with 01 to get the low bit only
"paddb   xmm1, xmm1;"     // shift left by 1 byte wise
"pxor    xmm2, xmm1;"
```

Algorithm 6 explains how to perform squaring in normal basis representation of GF(2^8). First, copy the content of a to b using the "movdqa" instruction, which copies sixteen bytes in parallel (Line 1). Then, Shift a left by one bit with the parallel add bytes "paddb" instruction that shifts all the sixteen bytes to the left concurrently (Line 2). Next, shift b right by 7 bits, but the bit-wise shift right operation is not supported in the existing instruction set. Therefore, we used the "psrlw" instruction to perform the shift right operation for the 16 bytes simultaneously followed by AND with the hexadecimal value 0x01 to get just the MSB through the parallel AND bytes "pand" instruction (Line 3). Finally, the result of the rotate left operation is accomplished in line (Line 4) by XORing a with b using the "pxor" instruction, which performs sixteen simultaneous XOR operations.

4.6 Efficient SubByte Inversion

One way to calculate the multiplicative inverse is based on the exponentiation operation using the addition chain as mentioned before to compute $A^{-1} = A^{254}$, which contains squares and multiplications. Our proposed idea is to start with the polynomial basis to perform the multiplications and whenever we face a square operation, we convert A from the polynomial to the normal basis using (21) then perform the square using our efficient normal basis squaring that was described in Section 4.5 and finally convert back to the polynomial basis using (22).

4.7 Integrate Secure Lookup Tables

As mentioned in [57], to mitigate cache attacks, the authors proposed avoiding the use of any lookup tables, which is one way. In our proposed technique, instead of avoiding the use of lookup tables we incorporated a Data-Oblivious Memory Access

Pattern technique, which is another way to mitigate cache attacks as described in [41]. The idea is that the memory access pattern must be completely oblivious to the data processed through the encryption process. To perform that, one reads all entries of the related lookup table, in fixed order, whenever any single one is needed.

In our case, we used four small lookup tables of 16 bytes each to speed up the basis conversion process. Then, whenever the conversion process is needed, the "movaps" instruction is used to load the entire table to one XMM register (each XMM register can hold 16 bytes). After that, perform 16 bytes table lookup at the same time utilizing the "pshufb" instruction. Accordingly, we performed 16 byte lookup tables in parallel without losing the security level against cache attacks.

4.8 Security Analysis

From [10] [13] [41], the security of our proposed technique can be easily proven. Moreover, we believe that the proposed countermeasure can protect AES algorithm from several SCA such as power analysis, electromagnetic, timing and cache attacks since we employ a higher-order masking scheme [13], achieving a constant-time implementation [10] (by using SIMD technology) and a data-oblivious memory access pattern [41].

4.9 Implementation Results

We implement this efficient Multiplicative Inverse by incorporating SSSE3 instructions with C-language. Rather than using the intrinsic functions that are easier to deal with, we chose inline assembly code for better performance. We replaced the multiplicative inverse of the S-box that was used in CHAPTER 2. We assumed that a pool of independent and uniformly distributed random masks was available securely and was unreachable by attackers.

Table 4. Comparing Encryption Execution Time and Size of Lookup Tables with Other Schemes

Scheme	**Cycles ($\times 10^3$cc)**	**Table size(Bytes)**	**Times**
	Encryption	**LUT**	**Speedup**
Original AES (Straightforward AES)			
	9.0	255	
First Order Masking			
[13]	129	3153	13.87
[57]	13	0	1.39
Ours	9.3	64	
Second Order Masking			
[13]	271	3845	13.89
[57]	29	0	1.49
Ours	19.5	64	
Third Order Masking			
[13]	470	4648	10.44
[57]	61.7	0	1.37
Ours	45	64	

For a fair comparison, we excluded the results of [52] and [58] because the S-box calculations were based on the composite field. We compared our implementation results with two different implementations in [13] and [57]. In [13], the higher-order masking scheme was implemented and the efficiency of the multiplication over GF(2^8) operation was improved using a log table. In [57], three countermeasures have been combined and the S-box calculations are over GF(2^8). The performance problem has been solved using SIMD technology. In our technique the square operation in S-box is computed over the normal basis GF(2^8) and efficient secure parallel lookup tables are used. However, as shown in Table 4, we gain a speedup of 13.87, 13.89 and 10.44 times faster than the existing countermeasure in [13] for first, second and third order masking, respectively. Our speedup is 1.39, 1.49 and 1.37 times faster than the existing countermeasure in [57] for first, second and third order masking.

CHAPTER 5

Case Study: AES Implementation in the OpenSSL Framework

As part of our mission we want to chose a useful framework that suffers from SCAs, then deploy our techniques to protect it with acceptable performance. Thus, we chose the AES implementation under the OpenSSL library as our case study.

5.1 OpenSSL

OpenSSL [59] is an open source cryptography library that provides a robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. It is also a general-purpose cryptography library that combines many cryptographic implementation protocols. It has found wide use in internet web servers, serving a majority of all web sites. It is writtien in the C programming language. It is free to download and to use for commercial and non-commercial purposes, under some license conditions [60]. For security purposes, it is important for the OpenSSL users (developers, vendors, or individual users) to know the supported versions that are eligible for future security fixes. Whenever a new type of threat succeeds in attacking the OpenSSL library, cryptographic engineers and OpenSSL developers work hard to fix that vulnerable part and release a new version. Then, it is recommended to update the library as soon as possible to protect confidential information from any dangerous leak. More details about the features of each version, supported and unsupported versions are available in [61].

Program developers usually use the OpenSSL library to add strong cryptography features to their programs. On the other hand, it is also a powerful command line tool and can be used by shell scripts [62]. OpenSSL binary affords command-line access to many cryptographic operations and applications such as:

- openssl dgst – to produce a digest of a file using hash functions or digital signature algorithms.
- openssl enc – to encrypt or decrypt data using various symmetric block and stream ciphers.
- openssl speed – to test the performance of any supported cryptographic algorithms.
- openssl s_client – a TCP and TLS client, which is able to connect to a remote host.
- openssl s_time – a client which benchmarks the performance of a TLS connection.

Moreover, OpenSSL library supports many cryptographic algorithms such as:

- openssl list-message-digest-algorithms.
- openssl list-cipher-algorithms.
- openssl list-public-key-algorithms.

5.2 Motivations

The Intel Atom processor is part of Intel's x86-64 family, which supports the SSSE3 optimization instruction set and hence consumes low power. These processors are widely used in everyday small devices such as smartphones and netbooks.

In [14] the authors report that they mounted an electromagnetic SCA on the OpenSSL implementation of the AES algorithm and were able to retrieve part of the secret key. Their project used an Intel Atom processor. Moreover, in [16] the authors successfully mounted a cache timing SCA on OpenSSL implementation of the AES algorithm they were able to recover the full secret key. They targeted a Pentium III processor. In [8] the authors succeed in performing electromagnetic attacks on modern laptop and desktop systems with various Intel processors. Those issues inspired us to produce a combined countermeasure to mitigate different types of attack using Intel SSSE3 instructions in modern Intel processors, and any other processor that supports SIMD technology.

Our techneques are investigated in terms of performance metrics which in our case are the number of clock cycles (execution time), table size (area overhead) and speedup. To verify our work, we implemented all the discussed countermeasures of AES by incorporating SSSE3 instructions with C language and incorporated them under the OpenSSL library in a Linux machine. Moreover, instead of using the intrinsic functions (to levarage SSSE3 instructions with C code) that are easier to deal with, we chose inline assembly code for better performance but that required us to manage register usage and do instruction scheduling.

CHAPTER 6

Conclusions

In CHAPTER 2 we addressed the problem of producing, for AES, a leakage free countermeasure against multiple side channel attacks (differential power analysis, electromagnetic, timing and cache attacks) at the same time maintaining a reasonable and practical speed, by combining more than one countermeasure (high order masking scheme, constant time implementation, and avoidance of lookup tables) to mitigate more than one type of SCA. Then, we solved the penalty on performance as a result of combining those countermeasures by implementing our scheme using SIMD technology (to gain the advantage of data-level parallelism). The speed of our implementation makes it more practical for use by the real world applications and hence attractive.

In CHAPTER 3 we addressed the problem of increasing the performance of the bottleneck transformation (SubByte transformation) while maintaining the same level of security, by apply enhancement techniques in three different levels:

1. Algorithm Level: evaluating the SubByte using a composite field, where carrying out the computations over GF(2^4) is much faster that over GF(2^8). Moreover, accomplishing multiple operations while incurring only the cost of one.

2. Byte Level: Take advantage of the parallel computing by using Single Instruction Multiple Data (SIMD) technology to achieve data level parallelism.
3. Bit Level: Deriving the formulas for each bit of each operation (if applicable), to minimize the amount of required calculations.

In CHAPTER 4 we addressed the problem of increasing the performance of the first operation in the bottleneck transformation (SubByte transformation) the multiplicative inverse operation with the same level of security by:

1. Utilizing the squaring property of the normal basis [55], where the square in normal basis is just a cyclic shift left.
2. Solving the basis conversion problem and minimizing the previously required extensive calculations using Karnaugh maps [54] and Boolean algebra.
3. Utilizing SIMD technology, which can manipulate 16 bytes concurrently, to achieve data level parallelism [56].
4. Taking advantage of the "pshufb" instruction to design an enhanced countermeasure against cache attacks, which performs Data-Oblivious Memory Access Pattern technique instead of Avoid Using any Lookup Tables technique.

In CHAPTER 5 we verify our work by applying our techniques in the OpenSSL implementation of the AES algorithm as a real world, very important and useful application.

CHAPTER 7

References

[1] J. Daemen and V. Rijmen, in *The design of Rijndael: AES-the advanced encryption standard*, Springer Science & Business Media, 2013.

[2] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Computer Networks 48, no. 5*, 2005, pp. 701-716.

[3] J. Daemen and V. Rijmen, "The Design of Rijndael: AES-the advanced encryption standard," *Springer Science & Business Media,* 2013.

[4] N. Selvaraju and G. Sekar, "A method to improve the security level of ATM banking systems using AES algorithm," *International Journal of Computer Applications 3, no. 6,* 2010.

[5] E. Brier, C. Clavier and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *Cryptographic Hardware and Embedded Systems-CHES 2004*, Springer Berlin Heidelberg, 2004, pp. 16-29.

[6] P. Kocher, J. Joshua , J. Benjamin and . R. Pankaj, "Introduction to differential

power analysis," in *Journal of Cryptographic Engineering 1, no. 1*, 2011.

[7] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO'96*, Springer Berlin Heidelberg, 1996, pp. 104-113.

[8] A. Zajic and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," in *Electromagnetic Compatibility, IEEE Transactions on 56, no. 4*, 2014.

[9] E. Brickell, G. Graunke, M. Neve and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," in *IACR Cryptology ePrint Archive 2006*.

[10] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *Cryptographic Hardware and Embedded Systems-CHES 2009*, Springer Berlin Heidelberg, 2009, pp. 1-17.

[11] C. Carlet, L. Goubin, E. Prouff, M. Quisquater and M. Rivain, "Higher-order masking schemes for s-boxes," in *Fast Software Encryption*, Springer Berlin Heidelberg, 2012, pp. 366-384.

[12] J.-S. Coron, "Higher order masking of look-up tables," in *Advances in Cryptology–EUROCRYPT 2014*, Springer Berlin Heidelberg, 2014, pp. 441-458.

[13] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, Springer Berlin Heidelberg, 2010, pp. 413-427.

[14] A. Do, S. Thet Ko and A. Thu Htet, "Electromagnetic side-channel analysis on Intel Atom Processor," Worcester Polytechnic Institute, 2013.

[15] J. Viega, M. Messier and P. Chandra, "Network Security with OpenSSL," in *Cryptography for Secure Communications*, O'Reilly Media, Inc.", 2002.

[16] H. Aly and M. ElGayyar, "Attacking AES using Bernstein's attack on modern processors," in *Progress in Cryptology–AFRICACRYPT 2013*, Springer Berlin Heidelberg, 2013, pp. 127-139.

[17] J. Katz and Y. Lindell, in *Introduction to modern cryptography*, CRC Press, 2014.

[18] P. Gallagher, "Secure hash standard (SHS)," FIPS PUB (2012): 180-4.

[19] D. Eastlake 3rd and P. Jones, US secure hash algorithm 1 (SHA1), No. RFC 3174, 2001.

[20] B. Kaliski, The MD2 message-digest algorithm, No. RFC 1319, 1992.

[21] R. Rivest, The MD4 message-digest algorithm, 1992.

[22] R. Rivest, The MD5 message-digest algorithm, 1992.

[23] H. Dobbertin, A. Bosselaers and B. Preneel, "RIPEMD-160: A strengthened version of RIPEMD," in *International Workshop on Fast Software Encryption*, Springer Berlin Heidelberg, 1996, pp. 71-82.

[24] R. A. Rueppel, Analysis and design of stream ciphers, Springer Science & Business Media, 2012.

[25] R. Rivest, "A Description of the RC2 (r) Encryption Algorithm," 1998.

[26] R. Baldwin and R. Rivest, "The rc5, rc5-cbc, rc5-cbc-pad, and rc5-cts algorithms," No. RFC 2040, 1996.

[27] R. L. Rivest, M. J. B. Robshaw, R. Sidney and Y. Lisa Yin, "The RC6TM block cipher," in *First Advanced Encryption Standard (AES) Conference*, 1998.

[28] C. De Canniere, A. Biryukov and B. Preneel, "An introduction to block cipher cryptanalysis," in *Proceedings of the IEEE 94, no. 2*, 2006, pp. 346-356.

[29] E. Biham and A. Shamir, Differential cryptanalysis of the data encryption standard, Springer Science & Business Media, 2012.

[30] D. Coppersmith, "The Data Encryption Standard (DES) and its strength against attacks," in *IBM journal of research and development 38, no. 3*, 1994, pp. 243-250.

[31] J. Daemen and V. Rijmen, "The block cipher Rijndael," in *International Conference on Smart Card Research and Advanced Applications*, Springer Berlin Heidelberg, 1998, pp. 277-284.

[32] J. Daemen and V. Rijmen, "Rijndael, the advanced encryption standard," *Dr. Dobb's Journal 26,* vol. 3, pp. 137-139, 2001.

[33] M. Dworkin, "Recommendation for block cipher modes of operation. methods and techniques," No. NIST-SP-800-38A. NATIONAL INST OF STANDARDS AND TECHNOLOGY GAITHERSBURG MD COMPUTER SECURITY DIV, 2001.

[34] R. L. Rivest, A. Shamir and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," in *Communications of the ACM 21, no. 2*, 1978, pp. 120-126.

[35] D. W. Kravitz, "Digital signature algorithm," in *U.S. Patent 5,231,668*, issued July 27, 1993.

[36] S. Mangard, E. Oswald and T. Popp, "Power analysis attacks: Revealing the secrets of smart cards," Vol. 31. Springer Science & Business Media, 2008.

[37] D. Boneh, R. A. DeMillo and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," *International Conference on the Theory and Applications of Cryptographic Techniques,* no. Springer Berlin Heidelberg, pp. 37-

51, 1997.

[38] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual International Cryptology Conference*, Springer Berlin Heidelberg, 1997, pp. 513-525.

[39] H. Handschuh and H. M. Heys, "A timing attack on RC5," in *International Workshop on Selected Areas in Cryptography*, Springer Berlin Heidelberg, pp. 306-318.

[40] D. Agrawal, B. Archambeault, J. R. Rao and P. Rohatgi, "The EM side—Channel (s)," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer Berlin Heidelberg, 2002, pp. 29-45.

[41] A. D. Osvik, A. Shamir and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Topics in Cryptology–CT-RSA 2006*, Springer Berlin Heidelberg, 2006, pp. 1-20.

[42] C. Rebeiro, D. Mukhopadhyay and S. Bhattacharya, Timing Channels in Cryptography: A Micro-architectural Perspective, Springer, 2014.

[43] P. Kocher, J. Jaffe and B. Jun, "Differential power analysis," in *Annual International Cryptology Conference*, Springer Berlin Heidelberg, 1999, pp. 388-397.

[44] R. Bevan and E. Knudsen, "Ways to enhance differential power analysis," in

International Conference on Information Security and Cryptology, Springer Berlin Heidelberg, 2002, pp. 327-342.

[45] E. Brier, C. Clavier and F. Olivier, "Optimal Statistical Power Analysis," in *IACR Cryptology ePrint Archive 2003*, p. 152.

[46] C. Clavier, J.-S. Coron and N. Dabbous, "Differential power analysis in the presence of hardware countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer Berlin Heidelberg, 2000, pp. 252-263.

[47] J.-S. Coron and I. Kizhvatov, "Analysis and improvement of the random delay countermeasure of CHES 2009," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Springer Berlin Heidelberg, 2010, pp. 95-109.

[48] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof and F.-X. Standaert, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer Berlin Heidelberg, 2012, pp. 740-757.

[49] J. Daemen and V. Rijmen, The Design of Rijndael: AES-the advanced encryption standard, Springer Science & Business Media, 2013.

[50] Y. Ishai, A. Sahai and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology-CRYPTO 2003*, Springer Berlin

Heidelberg, 2003, pp. 463-481.

[51] H. Kim, T. H. Kim, D.-G. Han and S. Hong, "Efficient Masking Methods Appropriate for the Block Ciphers ARIA and AES," *ETRI Journal 32,* vol. 3, pp. 370-379, 2010.

[52] H. Kim, S. Hong and J. Lim, "A fast and provably secure higher-order masking of AES S-box," in *Cryptographic Hardware and Embedded Systems–CHES 2011*, Springer Berlin Heidelberg, 2011, pp. 95-107.

[53] A. Satoh, S. Morioka, K. Takano and S. Munetoh, "A compact Rijndael hardware architecture with S-box optimization," in *International Conference on the Theory and Application of Cryptology and Information Security*, Springer Berlin Heidelberg, 2001, pp. 239-254.

[54] S. D. Brown, "Fundamentals of digital logic with Verilog design," Tata McGraw-Hill Education, 2007.

[55] R. Lidl and H. Niederreiter, " Introduction to finite fields and their applications," Cambridge university press, 1994.

[56] H. Jeong, S. Kim, W. Lee and S.-H. Myung, "Performance of SSE and AVX instruction sets," arXiv preprint arXiv:1211.0820 (2012).

[57] A. Miyajan, Z. Shi, C.-H. Huang and T. F. Al-Somani, "An efficient high-order

masking of AES using SIMD," in *Computer Engineering & Systems (ICCES), 2015 Tenth International Conference*, IEEE, 2015, pp. 363-368.

[58] A. Miyajan, Z. Shi, C.-H. Huang and T. F. Al-Somani, "Accelerating higher-order masking of AES using composite field and SIMD," 2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT), 2015, pp. 575-580.

[59] "https://www.openssl.org".

[60] "https://www.openssl.org/source/license.html".

[61] "https://www.openssl.org/policies/releasestrat.html".

[62] P. Chandra, M. Messier and J. Viega, "Network security with OpenSSL," no. O'Reily, 2002.

[63] A. S. Coronado, "Computer Security: Principles and Practice," *Journal of Information Privacy and Security 9 no. 2,* pp. 42, 57, 2013.

[64] W. Alan, "[image] Available at : https://www.profwoodward.org/2012/01/emerging-threat-to-public-key.html," 2012.

[65] H. Alexander , "http://alexander.holbreich.org/symmetric-key-cryptography/".

[66] "https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation".

[67] FIPS PUB 197, "Advanced Encryption Standard (AES), National Institute of Standards and Technology, U.S. Department of Commerce," November 2001. Link in: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[68] A. Satoh, S. Morioka, K. Takano and S. Munetoh, "A compact Rijndael hardware architecture with S-box optimization," in *Advances in Cryptology—ASIACRYPT 2001*, Springer Berlin Heidelberg, 2001, pp. 239-254.

[69] J. Wolkerstorfer, E. Oswald and M. Lamberger, "An ASIC implementation of the AES SBoxes," in *Topics in Cryptology—CT-RSA 2002*, Springer Berlin Heidelberg, 2002, pp. 67-78.

[70] J. López and R. Dahab, "High-speed software multiplication in F2m," in *Progress in Cryptology—INDOCRYPT 2000*, Springer Berlin Heidelberg, 2000, pp. 203-212.

[71] E. Brickell, G. Graunke, M. Neve and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities," in *IACR Cryptology ePrint Archive 2006*, 2006.