A fail-fast mechanism for authenticated encryption schemes

Student Name: Naina Gupta

IIIT-D-MTech-CS-IS
December, 2015

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Dr. Donghoon Chang, IIIT Delhi (Chair)
Dr. Gaurav Gupta (Deity, Govt of India)
Dr. Goutam Paul (ISI Kolkata)

Submitted in partial fulfilment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Information Security

Certificate

This is to certify that the thesis titled **"A fail-fast mechanism for authenticated encryption schemes"** submitted by **Naina Gupta** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by her under our guidance and supervision in the Security and Privacy group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Donghoon Chang
Indraprastha Institute of Information Technology, New Delhi

Abstract

In the modern world, almost every computing device uses some cryptographic technique or the other. Over the years several schemes have been proposed implemented and standardized. For any kind of data transfer the primary goals are encryption and authentication. Historically, these two goals are achieved separately, via two different techniques. Any symmetric cipher scheme can be used for encryption, whereas, for authentication, usage of a keyed MAC is prevalent. There is another approach known as Authenticated Encryption (AE), which fulfills both the goals at the same time.

From an implementation perspective, it is important that, if the packet is malformed, it is rejected as soon as possible. Common techniques like AES-CBC, allow for such a fail-fast paradigm using padding oracle. But, the same technique cannot be applied for other common AE techniques like AES-GCM. In this work, we provide a technique using which any AE scheme can be used directly (without any change), whilst providing the good fail-fast features at the same time.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr.Donghoon Chang for his guidance and support. The quality of this work would not have been nearly as high without his well-appreciated advice. I would like to extend my gratitude to Arpan Jati for devoting his time in discussing the ideas, helping me in learning the various device working and running the code, and giving his invaluable feedback. I am grateful to him for providing me the guidance whenever needed for this research. I appreciate the support Megha Agrawal and Amit Kr. Chauhan has provided me in writing the security analysis of my work. I would like to dedicate this thesis to my loving and supportive parents who have always been with me, no matter where I am.

I thank Dr. Gaurav Gupta and Dr. Goutam Paul for accepting to be a part of my thesis committee as the internal examiner and as the external examiner, and for enriching this thesis with their valuable suggestions and feedback.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we outline the primary motivation of this research work and describe the contributions.

1.1 Approaches for securing data in transit

In the growing world of technology, the amount of information we share across the network is constantly growing day by day. There is a need to conceal confidential information. Confidentiality and Integrity are two main factors which are considered in order to protect a message. There are two main approaches to achieve the said goals:

1. The first approach is to treat encryption and authentication separately. A block cipher or stream cipher is used to encrypt the message and a hash function is used to generate the MAC of the message as shown in 1.1. For example, we can use Advanced Encryption Standard (AES) [4] for encrypting the message and Hash Message Authentication Code (HMAC) [10] to generate the hash of the message (MAC tag). In this approach, the sender encrypts the message using a K_1 and authenticates it using K_2 and sends the encrypted message appended with the MAC to the receiver. At the receiver side, message if first decrypted using K_1 and then generates the hash using K_2. It then verifies the generated MAC with the MAC received from the sender, if it matches then the packet is accepted otherwise it is rejected straightforward.

2. Another approach is use an integrated Authenticated Encryption (AE) algorithm like AES-GCM for both message encryption and authentication. It is also more efficient than the first approach as it can share

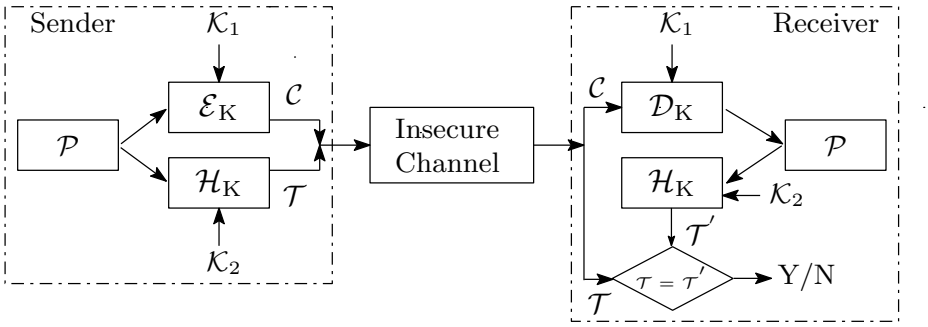

Figure 1.1: Using two seperate algorithms for encrypt and authenticate

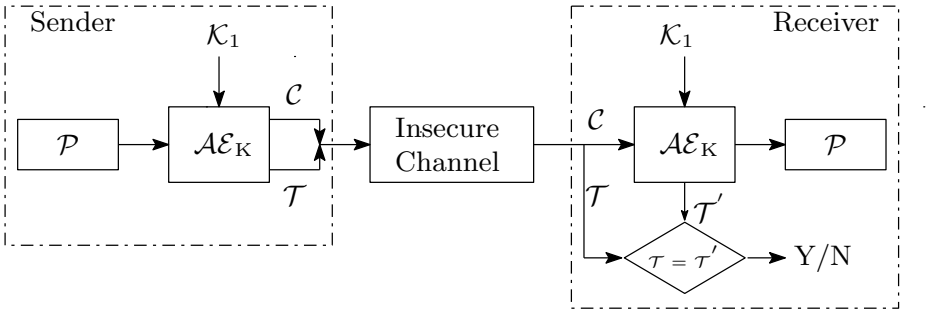

Figure 1.2: Using AE scheme

a part of the computation. Such kind of algorithms use only one key as shown in 1.2. Therefore, key exchange and storage issues are better as compared to using two different algorithms.

There are now two NIST recommended modes of operation for authenticated encryption, namely, Counter with Cipher Block Chaining Mode (CCM) [9] and Galois Counter Mode (GCM) [11].

1.2 Motivation

With the advent of ubiquitous secure protocols such as Secure Shell (SSH) [14] and Secure Sockets Layer / Transport Layer Security (SSL/TLS) [5], applications like WinZip, Disk Encryption, Password encryption and various devices like IoT's, embedded systems etc, sensitive data is increasingly transmitted through or stored on the systems using these applications. Users concerned about security typically employ protective methods to secure their data.

Algorithms which use MtE (MAC then Encrypt), have a fail-fast mechanism or padding oracle which discards the packet immediately based on the correctness of the padding. But, due to well known padding oracle attacks on this technique, new approaches like AE schemes were developed. AE schemes are being widely used by many servers and has a great scope of use in various applications and embedded devices in near future as well, but this scheme lacks a fail-fast mechanism.

A lot of processing power is spent for performing cryptographic operations in a typical server workload. Special instructions for many popular AE schemes are available in modern CPUs. These instructions offer tremendous speed-up but it still costs cycles to encrypt, decrypt and verify the integrity of a message; it takes more clock cycles as the message size is increased because existing AE schemes process the MAC tag at the end of the packet. Environments where available resources are restricted, these scheme will have a weakness of taking up a considerable amount of time to check the integrity of message.

For example, in WinZip if a wrong password is provided to the application during decryption, it will give warning only after processing the whole file. Thus, there arises a need for a new approach which can overcome this problem. In this thesis we explore the idea to use the current secure AE algorithms to lower the CPU load and possibly speed-up the message integrity verification by proposing a fail-fast mechanism. So, using the proposed approach, even if a wrong password is provided or a garbage packet is transmitted over the network, it is possible to detect and discard the packet at an early stage rather than processing the whole packet first and then discarding it.

1.3 Contribution

This thesis deals with the design and implementation of a generic algorithm which enriches the existing AE schemes by adding a fail-fast mechanism. We provide the description and implementation of both the existing AE scheme and the new design approach using an additional block to expedite the MAC verification process in the existing AE algorithms.

This work is an attempt to devise a new model which is both secure and fast. Our approach requires an extra block to be generated which is appended to the original ciphertext. The block generated is composed of parameters related to the packet so it can only be created by a legitimate person in possession of the key. Thus, ensuring an intruder cannot generate a

packet which has a valid initial block appended with invalid ciphertext/MAC tag. Server will immediately discard the packet if such a case is found.

Chapter 2

Background

In this chapter, we provide detailed explanation of how AES-CBC algorithm works, discuss about historical background of padding oracle attacks, and point out why they are still relevant. Subsequently, we will introduce the reader to the most basic cryptographic primitives and concepts required to understand padding oracle attacks. Then we continue with explaining padding oracle attacks in detail. Finally, we conclude with the limitations of AES-CBC algorithm for encryption/decryption.

2.1 History and Relevance

A practical padding oracle attack for symmetric cryptography has first been proposed by Vaudenay in 2002 [13]. Similar attacks, however, had already been shown theoretically feasible as early as 1998 for RSA [2], though not entirely as efficient. Thus, padding oracle attacks are known for more than a decade.
Still, there are standard and implementation errors which results in the emergence of such attacks. The main reason for such attacks is the MAC-then-Pad-then-Encrypt paradigm which they follow and is used by many, and thus cannot easily be fixed. As a consequence, still today, relevance is given to such attacks.

2.2 Notations used

Table 2.1 summarizes the notation used in this chapter.

Table 2.1: Notations used in background

Symbol	Meaning
$\{0,1\}^{8n}$	Bitstring consisting of 0,1 of length $8n$
$\oplus$	Bitwise XOR operation
$\mathcal{IV}$	Unique initialization vector
$\mathcal{K}$	Distinct Key of length k bits
b	block size, depending on the algorithm
$len(x)$	returns length of message x in bytes
$\mathcal{M}$	Plaintext message of length $\{0,1\}^{8n}$
$\mathcal{M}^{'}$	Plaintext message obtained after decryption
$\mathcal{C}$	Ciphertext message, length is same as of Plaintext
$\mathcal{C}^{'}$	Received ciphertext
$\mathcal{E}_{\mathcal{K}}$	A symmetric key encryption function
$\mathcal{D}_{\mathcal{K}}$	A symmetric key decryption function
Encipher/Encrypt	interchangeable terms for encryption
Decipher/Decrypt	interchangeable terms for decryption

2.3 Cryptographic Basics

In the following section, we give an overview of the basics of cryptographic algorithms and concepts related to AES-CBC which are further required to understand the padding oracle attacks. Also, we will explain how the basic encryption and decryption algorithm works.
Basic encryption and decryption functions can be represented by following equations :

$$\mathcal{E}_{\mathcal{K}}(\mathcal{M}) = \mathcal{C} \qquad //\text{Encryption Function}$$

and

$$\mathcal{D}_{\mathcal{K}}(\mathcal{C}') = \mathcal{M}' \qquad //\text{Decryption Function}$$

If $\mathcal{C} = \mathcal{C}' \Rightarrow \mathcal{M} = \mathcal{M}'$, else ciphertext has been tampered in transit.

Symmetric cipher types : Cryptographic algorithms are generally referred to as ciphers. There are two basic types of symmetric algorithms: block ciphers and stream ciphers. Block ciphers operate on blocks of plaintext and ciphertext at a time whereas Stream ciphers operate on streams of plaintext and ciphertext one bit or byte at a time.
A cryptographic mode usually combines the basic cipher, some sort of feedback, and some simple operations.

2.3.1 Padding

Most messages dont divide neatly into the block size as specified by the cryptographic algorithm being used as the input may come in varying sizes, there is usually a short block as the end. Padding is the way to deal with this problem. Padding allows to append the last block with some regular pattern in order to make the block aligned in size. This has to be done in such a way that it becomes distinguishable from the true payload so that it can be easily removed at decryption time. There are various standards which describe different methods of how padding can be performed. For example, the Public Key Cryptography Standard (PKCS)#7 [8] defines that each added byte should be the value of the number of bytes to be added in padding, i.e., if there need to be 5 bytes added, each of these bytes will have the value 0x05. Even messages being a multiple of the block size need to be padded. In this case, one full block of padding with bytes of value b are appended, with b being the block size. Thus, at least the last byte of a message needs to be padding and padding must always be present.

2.3.2 Cipher Block Chaining

Block ciphers can operate in different modes. These modes of operation have been devised to encipher/decipher text of any size by breaking into chunks of data (known as blocks). Further, they offer different services like confidentiality, integrity and authenticity.
Cipher Block Chaining (CBC) is the most common mode of block cipher being used these days because of its chaining mechanism. Chaining adds a feedback mechanism to the block cipher. The results of encryption of previous blocks are fed into the encryption of the current block, thus making each ciphertext block to be dependent on the plaintext that generated it as well as on all the previous plaintext blocks.
Below algorithms 1 and 2 describes how encryption and decryption works in AES-CBC. Also refer Figure 2.1 for more details.

Algorithm 1: CBC Encryption

AES-CBC-E$_{\mathcal{K}}(\mathcal{IV}, \mathcal{M})$

Input : Key K;
Initialization Vector $\mathcal{IV}$(denoted as $\mathcal{C}_0$);
Plaintext message $\mathcal{M}$

Output: Ciphertext $\mathcal{C}$

Add padding to the plaintext message $\mathcal{M}$ so that it becomes a mutiple of block size i.e. 16 bytes in AES.

Let n (no of blocks) = $\frac{len(\mathcal{M}\|Padding)}{16}$

for $i \leftarrow 1$ **to** n **do**

    temp = $\mathcal{M}_{\text{i}} \oplus \mathcal{C}_{\text{i-1}}$;

    $\mathcal{C}_{\text{i}} = \mathcal{E}_{\text{K}}(temp)$;

end

return ciphertext $\mathcal{C}$;

Algorithm 2: CBC Decryption

AES-CBC-D$_{\mathcal{K}}(\mathcal{IV}, \mathcal{C})$

Input : Key K;
Initialization Vector $\mathcal{IV}$(denoted as $\mathcal{C}_0$);
Ciphertext $\mathcal{C}$

Output: Plaintext message $\mathcal{M}$

Let n (no of blocks) = $\frac{len(\mathcal{C})}{16}$

for $i \leftarrow 1$ **to** n **do**

    temp = $\mathcal{D}_{\text{K}}(\mathcal{C}_{\text{i}})$;

    $\mathcal{M}_{\text{i}} = temp \oplus \mathcal{C}_{\text{i-1}}$;

end

remove padding from the decrypted message to get the actual payload;

return plaintext $\mathcal{M}$;

2.4 Padding Oracle Attacks

As discussed in section 1.2.2 block cipher modes requires a padding method to be used in order to make the block aligned to the block size b before encryption.
Standard-conforming padding, in addition to the design principles behind

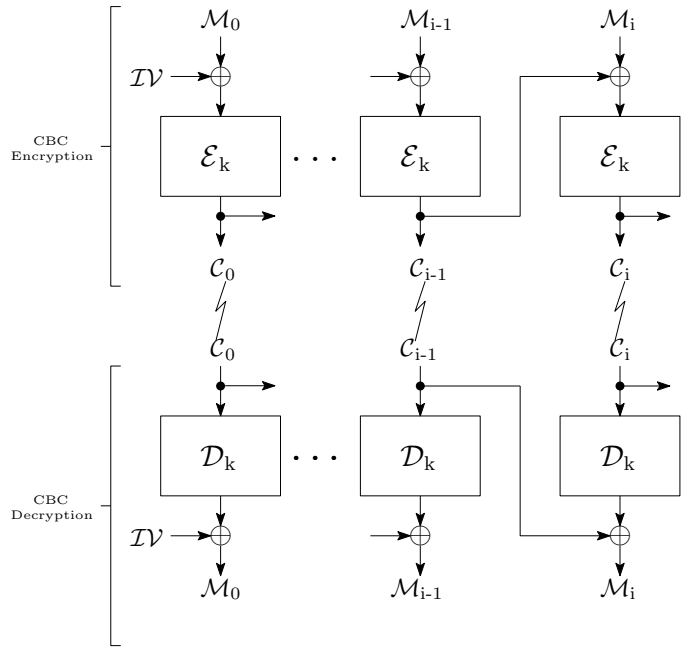

Figure 2.1: CBC mode encryption and decryption

most implementations of cryptographic standards, make portions of the plaintext easily guessable. One correctly guessed byte at the end of a message tells an adversary the value of a number of other bytes, with the number of bytes determined by the guessed bytes value. This is an important factor in the feasibility of padding oracle attacks.

In this section we will describe how padding oracle attacks works based on the properties of standard padding method PKCS #7(as shown below). This attack is not specifically on this padding method but applies to any padding scheme.

The functions PadLen(l), PadStr(l) and ValidPad(x) shown below describes how padding length is determined, how the padding string which needs to be appended is calculated conforming to (PKCS)#7 standard and then how it is verified whether a message x has valid padding or not.

Algorithm 3: Calculate the padding length

```
1 PadLen(l)
  Input  : length l of message
           in bytes
  Output: padding length
2 return 16 − (l % l6)
```

Algorithm 4: Calculate the padding bytes string

```
1 PadStr(l)
  Input  : padding length
           denoted by l
  Output: padding string of
           length l to be
           appended at end of
           message
2 return l , ... , l , l
```

Algorithm 5: Check for valid/invalid padding

```
1  ValidPad(x)
   Input  : message x
   Output: returns true/false
           depending on
           whether x has
           valid/invalid
           padding
2  if | x | %16 ≠ 0 then
3  |  return false;
4  else
5  |  l := last byte of x ;
6  |  if l ∉ {1, ..., 16} then
7  |  |  return false;
8  |  else
9  |  |  if PadStr(l) = last l
   |  |   bytes of x then
10 |  |  |  return true;
11 |  |  else
12 |  |  |  return false;
13 |  |  end
14 |  end
15 end
```

2.4.1 The setting.

When a server receives the ciphertext , it first decrypts the ciphertext and checks whether the plaintext has valid padding or not depending on the padding method agreed by both sides.
The server behaviour changes depending on whether the padding is valid and it can be observed easily as server gives a special error message in the case of invalid padding (add reference for different error messages padding oracle attack). Also, the difference in response time when processing a ciphertext with invalid padding is enough to allow the attack to work. (add

reference for timing padding oracle attack). For this attack to work we will assume, that there exists a padding oracle which provides the information whether a ciphertext has a valid padding or not to the attacker. It does not matter how does attacker gains this information but he has access to this padding oracle, which gives the same information as following function :

Algorithm 6: Padding oracle responds whether input has valid padding or not

```
1 PaddingOracle(C)
  Input  : ciphertext C
  Output: returns true/false depending on valid/invalid padding
2 M = D_K(C) ;
3 return ValidPad(M);
```

In the later section we will describe, even if the server does not provide any information about the ciphertext to the attacker, still the attacker will be able to decrypt any ciphertext of its choice using the padding oracle.

2.4.2 Malleability of CBC encryption

As described in section 2.3.2 about CBC decryption, if the ciphertext is $\mathcal{C} = \mathcal{C}_0, \ldots, \mathcal{C}_\mathrm{n}$, then the i^{th} plaintext block is computed as :

$$\mathcal{M}_\mathrm{i} = \mathcal{D}_\mathcal{K}(\mathcal{C}_\mathrm{i}) \oplus \mathcal{C}_\mathrm{i\text{-}1}$$

From this we can deduce two important facts :

1. If we remove any two consecutive ciphertext blocks $(\mathcal{C}_\mathrm{i\text{-}1}, \mathcal{C}_\mathrm{i})$,then they are by themselves a valid encryption of $\mathcal{M}_\mathrm{i}$ from the definition of CBC mode encryption. Thus, providing the attacker an opportunity to focus on decrypting a single block at a time.

2. If we XOR $\mathcal{C}_\mathrm{i\text{-}1}$ with a known value x, the result $(\mathcal{C}_\mathrm{i\text{-}1} \oplus x,\ \mathcal{C}_\mathrm{i})$ is a ciphertext that decrypts to $\mathcal{M}_\mathrm{i} \oplus x$

$$\mathcal{D}_\mathcal{K}(\mathcal{C}_\mathrm{i\text{-}1} \oplus x,\ \mathcal{C}_\mathrm{i}) = \mathcal{D}_\mathcal{K}(\mathcal{C}_\mathrm{i}) \oplus (\mathcal{C}_\mathrm{i\text{-}1} \oplus x) = (\mathcal{D}_\mathcal{K}(\mathcal{C}_\mathrm{i}) \oplus \mathcal{C}_\mathrm{i\text{-}1}) \oplus x = \mathcal{M}_\mathrm{i} \oplus x$$

If we send such a ciphertext $(\mathcal{C}_\mathrm{i\text{-}1} \oplus x,\ \mathcal{C}_\mathrm{i})$ to the padding oracle, we would be able to get to know whether $\mathcal{M}_\mathrm{i} \oplus x$ is a (single block) with valid padding or not. By iterating over different values x and asking questions of this form

to the padding oracle, we can eventually learn all of $\mathcal{M}_{\text{i}}$.
The following function can be used to ask such questions from the padding oracle:

Algorithm 7: Checks whether the modified input $\mathcal{M}_{\text{i}} \oplus x$ has valid/invalid padding

```
1 CheckModInput(C, i, x)
  Input  : ciphertext M, block number i, guessed byte value x
  Output: returns true/false depending on valid/invalid padding
2 M = D_K(C) ;
3 return PaddingOracle(C_{i-1} ⊕ x, C_i);
```

2.4.3 Learning the last byte of a block.

We now show how to use the CheckModInput function to determine the last byte of a plaintext block $\mathcal{M}$.
This is the easy case, as shown below. We can try every possible value for byte x and ask to the padding oracle whether $\mathcal{M} \oplus x$ has valid padding or not. Since the last byte of $\mathcal{M}$ (and hence $\mathcal{M} \oplus x$) should be 01 for it to be a valid padding, only one of the possible value of x will give valid padding. Thus, there will be a unique candidate for x which results in $\mathcal{M} \oplus x$ with valid padding.

...	...	a0	42	m	m = unknown plaintext byte
$\oplus$				x	x = candidate byte
...	...	a0	42	01	valid padding $\Leftrightarrow x \oplus m = 01$

2.4.3.1 How it works.

Consider the function GuessLastByte and the scenario shown in Figure 2.2 where x is the byte we want to guess by making modifications in byte denoted by c.

Following function LearnLastByte summarizes the overall approach for

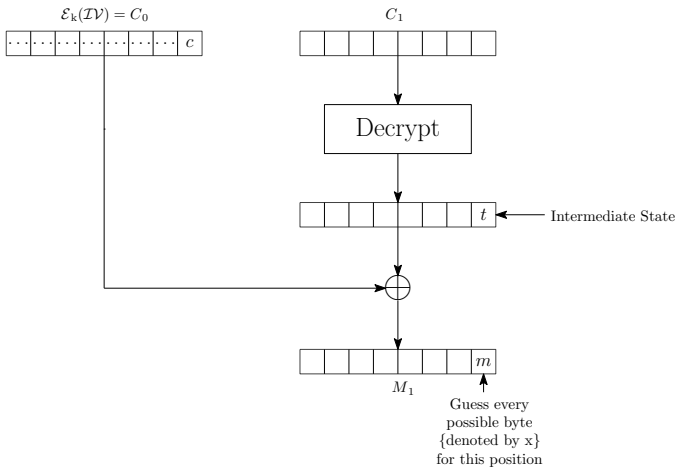

Figure 2.2: Scenario for attack working

Algorithm 8: Detailed explanation of guessing the last byte

GuessLastByte $(\mathcal{C}_{i-1}, \mathcal{C}_i)$

Known : $m = c \oplus t$

Description: Make guesses for x such that last byte of $\mathcal{M}$ becomes 01 for padding oracle to return "Valid Padding"

Input : Any two consecutive ciphertext block for example we considered C_0,C_1 in this case

Output : value of t (intermediate state last byte)

do

  modify $\mathcal{C}_0$ to $\mathcal{C}_0{}'$ such that last byte of $\mathcal{C}_0{}'$ becomes $= c \oplus x \oplus 01$

  Now m' becomes $= t \ \oplus$ (changed value of c)
  $= t \ \oplus \ c \ \oplus \ x \ \oplus \ 01$
  $= m \ \oplus \ x \ \oplus \ 01$ () (because $m = c \oplus t$)

  Now **if** *the guess is correct* **then**

    $m = x \Rightarrow$
    $m' = m \oplus m \oplus 01 = 01 =$ Valid padding

  else

    $m \neq x \Rightarrow$
    $m' = m \oplus x \oplus 01 =$ some value $\Rightarrow$ Invalid padding

  end

while *(guess for x returns Valid Padding)*;

guessing the last byte of the plaintext

Algorithm 9: learn the last byte of i^{th} plaintext block

```
1 LearnLastByte(C, i)
  Input  : ciphertext C, block number i
  Output: returns the last byte of plaintext block
2 for x ← 0 to 255 do
3 |   if CheckModInput(C, i, x) then
4 |   |   return x ⊕ 01 ;
5 |   else
6 |   |   continue ;
7 |   end
8 end
```

2.4.4 Learning other bytes of the block.

It is easy to learn other bytes of the block once we have learned the last byte of the block. Suppose we know the last 3 bytes of the plaintext block as shown in below example and now we want to figure out the 4^{th}-to-last byte. Since we know the last 3 bytes of $\mathcal{M}$, we can calculate a string s such that $\mathcal{M} \oplus s$ ends in 04 04 04 as for the block to have a valid padding, it should end with 04 04 04 04. So for the 4^{th}- to-last byte we can do the same thing as we did for the last byte, guessing the value for byte x such that $\mathcal{M} \oplus s$ results in a valid padding. Valid padding only occurs when the result has 04 in its 4^{th}-to-last byte.

	...	m	a0	42	3c	m = unknown plaintext byte
$\oplus$		04	04	04	04	p = PadStr(4)
$\oplus$			a0	42	3c	s = known bytes of $\mathcal{M}$
$\oplus$		x	00	00	00	y = candidate byte x shifted into place
	...	04	04	04	04	valid padding $\Leftrightarrow x = m$

Below algorithm explains the procedure to get the rightmost unknown byte

of $\mathcal{M}_i$.

Algorithm 10: learn rightmost unknown byte of $\mathcal{M}_i$, if we know that $\mathcal{M}_i$ ends in bytes s

```
1  LearnPrevByte(C, i, s)
   Input  : ciphertext C, block number i, M_i ends in bytes s
   Output: returns the rightmost unknown byte of plaintext block
2  p := PadStr(| s | + 1)
3  for x ← 0 to 255 do
4  |  y := x 00 ... 00 (where length of y = | s | + 1 s.t. first byte is x
   |    followed by 0's)
5  |  if CheckModInput(C, i, p ⊕ s ⊕ y) then
6  |  |  return x ;
7  |  else
8  |  |  continue;
9  |  end
10 end
```

2.4.5 Putting it all together.

The overall process of padding oracle attack to decrypt any ciphertext can be summarized as below:

Algorithm 11: learn the entire plaintext block $\mathcal{M}_i$

```
1  LearnBlock(C, i)
   Input  : ciphertext C, block number i
   Output: returns the entire plaintext block
2  s := LearnLastByte(C, i) ;
3  for x ← 0 to 15 do
4  |  b := LearnPrevByte(C,i,s) ;
5  |  s := b||s;
6  end
7  return s;
```

Algorithm 12: learn the entire plaintext $\mathcal{M}_1 \ \ldots \ \mathcal{M}_n$

```
1  LearnDec(C)
   Input  : ciphertext C
   Output: returns the complete plaintext message
2  M :=∈;
3  l := number of blocks in C, excluding IV;
4  for i ← 1 to l do
5  |  M := M||LearnBlock(C,i);
6  end
7  return M;
```

2.5 Limitations of AES-CBC

1. IV used in CBC mode must be unpredictable as mentioned by the NIST recommendation[[7],Section 5.3 and Appendix C], otherwise it may lead to various attacks. For example, BEAST([6]) attack demonstrated that if the attacker is able to predict the next IV to be used, then the attacker would be able shape the next chunk of plaintext.

2. Phil Rogaway in 1995 showed that if the IV chosen in not random for each plaintext message then there exists a statistical correlations between IV and plaintext such that it will leak some information to the attacker.

3. AES in CBC mode is a malleable cipher. Informally, an encryption scheme is malleable if an adversary can modify a ciphertext C in such a way to create a ciphertext C' whose plaintext P' is meaningfully related to original plaintext P. Because of this property of CBC mode, padding oracle attacks were possible as discussed in section 2.4.2.

4. Padding oracle attack on CBC is well known and was first introduced by Vaudenay [13]. He showed anyone in possession of "Valid padding oracle" can decrypt arbitrary ciphertext which were encrypted using Pad-then-CBC paradigm. This attack model was further enhanced by Paterson and Watson [12] who enriches the capablities of an adversary by providing an encryption oracle as well as ValidPadding oracle. So this will return the CBC-encryption of a properly padded plaintext if the plaintext has valid padding, otherwise it will return invalid ciphertext.
   Vaudenay focussed only on a single type of padding scheme. So Black and Urtubia [1] carried forward his idea and demonstrated that same idea of padding oracle attack can be applied to five more padding schemes. Canvel, Hiltgen, Vaudenay and Vuagnoux [3] further recovered SSL/TLS passwords using POA attack.

Chapter 3

Literature Review

In this chapter we focused on explaining the elements of GCM [11] and its associated notation. Firstly, the components (GHASH function and GCTR function) of GCM are discussed, followed by the high level structure of Authenticated Encryption and Decryption algorithm. Finally, the chapter is concluded with the discussion of limitations of GCM.

3.1 Notations used

Table 3.1 summarizes the notations used in this chapter.

3.2 AES-GCM Specifications

This section gives a detailed explanation of the AES-GCM algorithm.

3.2.1 Parameters and Components

GCM consists of mainly two functions authenticated encryption and authenticated decryption. Requirements and notations for input and output data of these two functions are discussed in following subsection. Also, GHASH and GCTR functions used for intermediate stages in GCM are discussed in detail in this section.

3.2.1.1 Authenticated Encryption

Following three inputs are required for authenticated encryption scheme:

- a key $\mathcal{K}$

Table 3.1: Notations used in literaure review

Symbol	Meaning
$\mathcal{IV}$	Initialization vector of length n = 1 to 2^{64} bits
$\mathcal{K}$	Distinct Key of length k bits
$\mathcal{A}$	Additional authenticated data of length between 0 to 2^{64}
$\mathcal{M}$	Plaintext message, can have any number of bits between 0 and $2^{39} - 256$
0^s	string consisting of $s^{'}0^{'}$ bits
$len(\mathcal{X})$	length of bit string $\mathcal{X}$
$\mathcal{X}_{\mathrm{i}} \| \mathcal{X}_{\mathrm{i+1}}$	concatenation of bit strings $\mathcal{X}_{\mathrm{i}}$ and $\mathcal{X}_{\mathrm{i+1}}$
$\mathcal{MSB}_{\mathrm{s}}(\mathcal{X})$	bit string consisting of s left-most bits of the bit string $\mathcal{X}$
$inc(\mathcal{X})$	output of GCM incrementing function applied to the block $\mathcal{X}$
$\mathcal{T}$	An authentication tag of length t which varies from 64 to 128 bits
$\mathcal{T}'$	Authentication tag obtained on receiver side after applying $\mathcal{AE}_{\mathcal{K}}$
$\mathcal{AE}_{\mathcal{K}}$	Authenticated Encryption function
$\mathcal{C}$	Ciphertext, length is same as of Plaintext

- an initialization vector $\mathcal{IV}$
- a datagram($\mathcal{A}$, $\mathcal{M}$) where
  - $\mathcal{A}$ is an additional authenticated data
  - $\mathcal{M}$ is the plaintext message

Following bit strings are the output of authenticated encryption function:

- $\mathcal{C}$ is the ciphertext(or encrypted payload)
- $\mathcal{T}$ is an authentication tag

In summary, it can be represented by following equations:

$$\mathcal{AE} : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^t$$

with

$$\mathcal{AE}(\mathcal{K}, \mathcal{IV}, \mathcal{A}, \mathcal{M}) = (\mathcal{C}, \mathcal{T})$$

3.2.1.2 Authenticated Decryption

Following parameters are taken as input :

- a key $\mathcal{K}$
- an initialization vector $\mathcal{IV}$
- a datagram($\mathcal{A}$, $\mathcal{C}$, $\mathcal{T}$) where
  - $\mathcal{A}$ is an additional authenticated data
  - $\mathcal{C}$ is the ciphertext(or encrypted payload)
  - $\mathcal{T}$ is an authentication tag

The output is one of the following:

- $\mathcal{M}$, decrypted plaintext message, if tag verification is successful
- $\perp$, failure if tag verification is failed

In summary, this can be represented by following equations:

$$\mathcal{AE} : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^t$$

with

$$\mathcal{AE}(\mathcal{K}, \mathcal{IV}, \mathcal{A}, \mathcal{C}) = \begin{cases} M, & \text{if } \mathcal{T}' \text{ is correct} \\ \perp, & \text{otherwise} \end{cases}$$

3.2.1.3 GHASH Function

GHASH function is used to provide the authentication mechanism in GCM. Input hash subkey $\mathcal{H}$, is obtained by applying the block cipher to the 128-bit string of all "0". The hash subkey $\mathcal{H}$ obtained is then multiplied to the input block $\mathcal{X}$, over a binary Galois field GF(2^{128}). Refer figure 3.1 and algorithm

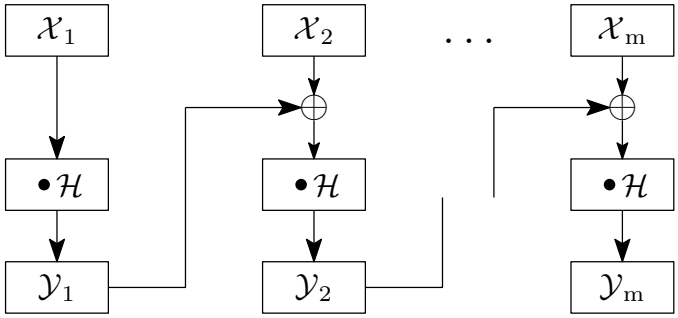

Figure 3.1: $\text{GHASH}_{\mathcal{H}}(\mathcal{X}_1 \| \mathcal{X}_2 \| \ldots \mathcal{X}_{m\text{-}1} \| \mathcal{X}_m) = \mathcal{Y}_m$

13 for detailed explanation of how this function works.

Algorithm 13: GHASH in AES-GCM

$\text{GHASH}_{\mathcal{H}}(\mathcal{X})$
Input : Bit string $\mathcal{X}$ of length s.t. it is a multiple of 128 bits;
Hash subkey $\mathcal{H}$
Output: Block $\mathcal{Y}_m$
Let m (no of blocks) $= \frac{len(\mathcal{X})}{128}$
$\mathcal{X} = \mathcal{X}_1 \| \mathcal{X}_2 \| \ldots \mathcal{X}_{m\text{-}1} \| \mathcal{X}_m$ representing the unique sequence of blocks
$\mathcal{Y}_0$ (a.k.a "zero block") = a bit string comprised by 128 binary 0
for $i \leftarrow 1$ **to** m **do**
    temp $= \mathcal{Y}_{i\text{-}1} \oplus \mathcal{X}_i$;
    $\mathcal{Y}_i = temp \bullet \mathcal{H}$;
end
return block $\mathcal{Y}_m$;

3.2.1.4 GCTR Function

Confidentiality mechanism of GCM is provided by GCTR function. The incrementing function, inc is used for generating the necessary sequence of counter blocks. First counter block is generated by incrementing the block derived from $\mathcal{IV}$. Refer figure 3.2 and algorithm 14 for detailed working of

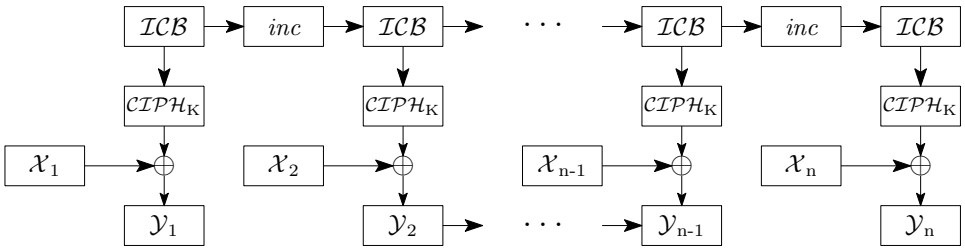

Figure 3.2: $\mathrm{GCTR}_{\mathcal{K}}(ICB, \mathcal{X}_1 \| \mathcal{X}_2 \| \ldots \mathcal{X}_{\mathrm{m\text{-}1}} \| \mathcal{X}_{\mathrm{m}}) = \mathcal{Y}_1 \| \mathcal{Y}_2 \| \ldots \mathcal{Y}_{\mathrm{m\text{-}1}} \| \mathcal{Y}_{\mathrm{m}}$

this function.

Algorithm 14: GCTR in AES-GCM

$\mathrm{GCTR}_{\mathcal{K}}(\mathcal{ICB}, \mathcal{X})$
Input : Bit string $\mathcal{X}$ of arbitrary length;
Initial counter block $\mathcal{ICB}$;
block cipher CIPH(such as AES);
Key $\mathcal{K}$
Output: Bit string $\mathcal{Y}$ of length same as $\mathcal{X}$
Let $m = \left\lceil \frac{len(\mathcal{X})}{128} \right\rceil$
$\mathcal{X} = \mathcal{X}_1 \| \mathcal{X}_2 \| \ldots \mathcal{X}_{\mathrm{m\text{-}1}} \| \mathcal{X}_{\mathrm{m}}$ representing the unique sequence of bit strings
Let $\mathcal{CB}_1 = \mathcal{ICB}$
for $i \leftarrow 2$ **to** m **do**
  $\mathcal{CB}_{\mathrm{i}} = inc(\mathcal{CB}_{\mathrm{i\text{-}1}})$
end
for $i \leftarrow 1$ **to** $m - 1$ **do**
  $\mathcal{Y}_{\mathrm{i}} = \mathcal{X}_{\mathrm{i}} \oplus CIPH_K(\mathcal{CB}_{\mathrm{i}})$
end
$\mathcal{Y}_{\mathrm{m}} = \mathcal{X}_{\mathrm{m}} \oplus MSB_{len(\mathcal{X}_{\mathrm{m}})}(CIPH_K(\mathcal{CB}_{\mathrm{m}}))$
Let $\mathcal{Y} = \mathcal{Y}_1 \| \mathcal{Y}_2 \| \ldots \mathcal{Y}_{\mathrm{m\text{-}1}} \| \mathcal{Y}_{\mathrm{m}}$
return $\mathcal{Y}$;

3.3 High level structure

This section discuss about how a message in encrypted and then decrypted using GCM algorithm. Block cipher used is AES.

3.3.1 Authenticated Encryption

Refer below algorithm 15 and figure 3.3 for step-by-step explanation of authenticated encryption function.

Algorithm 15: Authenticated Encryption function

AES-GCM-AE$_{\mathcal{K}}(\mathcal{IV}, \mathcal{M}, \mathcal{A})$

Input : Key $\mathcal{K}$;
block cipher CIPH(such as AES);
Tag length t;
Initialization vector $\mathcal{IV}$;
Plaintext message $\mathcal{M}$;
Additional authenticated data $\mathcal{A}$.

Output: Ciphertext message $\mathcal{C}$;
Authentication tag $\mathcal{T}$.

Let $\mathcal{H} = \mathcal{CIPH}_{\mathrm{K}}(0^{128})$

Define a block, $\mathcal{J}_0 = \mathcal{IV} \| 0^{31}1$ where $\mathcal{IV}$ is of length = 96 bits.

$\mathcal{C} = \mathcal{GCTR}_{\mathrm{K}}(inc(\mathcal{J}_0), \mathcal{M})$

Let $u = 128 \cdot \left\lceil \frac{len(\mathcal{C})}{128} \right\rceil - len(\mathcal{C})$ and let $v = 128 \cdot \left\lceil \frac{len(\mathcal{A})}{128} \right\rceil - len(\mathcal{A})$

Define block $\mathcal{S} = \mathcal{GHASH}_{\mathrm{H}}(\mathcal{A} \| 0^v \| \mathcal{C} \| 0^u \| [len(\mathcal{A})]_{64} \| [len(\mathcal{C})]_{64})$

$\mathcal{T} = \mathcal{MSB}_t(\mathcal{GCTR}_{\mathrm{K}}(\mathcal{J}_0, \mathcal{S}))$

return $(\mathcal{C}, \mathcal{T})$;

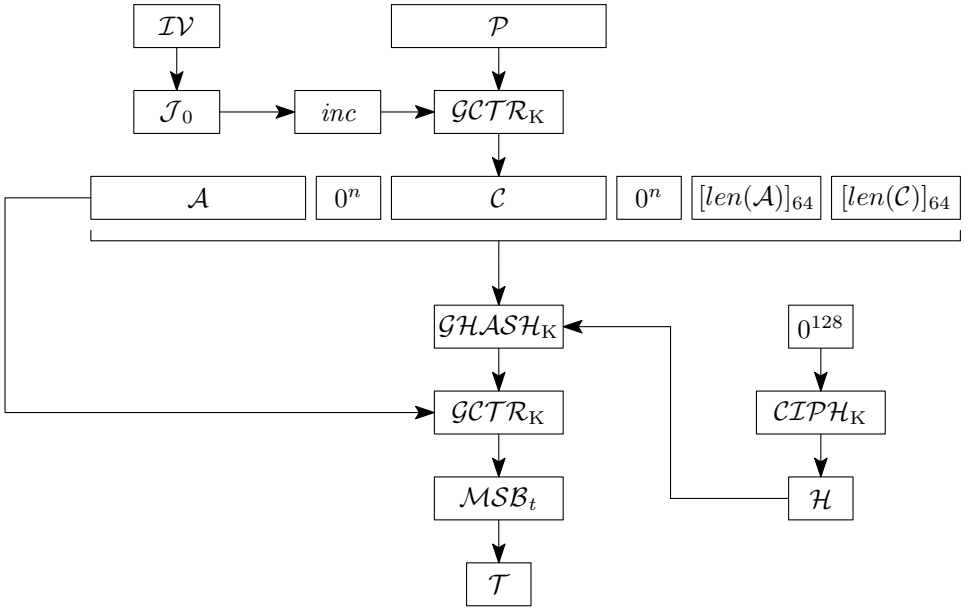

Figure 3.3: AES-GCM-AE$_{\mathcal{K}}(\mathcal{IV}, \mathcal{M}, \mathcal{A}) = (\mathcal{C}, \mathcal{T})$

3.3.2 Authenticated Decryption

Algorithm 16 and figure 3.4 explains the working of authenticated decryption function on GCM.

Algorithm 16: Authenticated Decryption function

AES-GCM-AD$_{\mathcal{K}}(\mathcal{IV}, \mathcal{C}, \mathcal{A}, \mathcal{T})$

Input : Key $\mathcal{K}$;
block cipher CIPH(such as AES);
Tag length t;
Initialization vector $\mathcal{IV}$;
Ciphertext message $\mathcal{C}$;
Additional authenticated data $\mathcal{A}$;
Authentication tag $\mathcal{T}$.

Output: Plaintext message $\mathcal{M}$ or a failure message.

Let $\mathcal{H} = \mathcal{CIPH}_{\mathrm{K}}(0^{128})$
Define a block, $\mathcal{J}_0 = \mathcal{IV} \| 0^{31}1$ where $\mathcal{IV}$ is of length = 96 bits.
$\mathcal{M} = \mathcal{GCTR}_{\mathrm{K}}(inc(\mathcal{J}_0), \mathcal{C})$
Let $u = 128 \cdot \left\lceil \frac{len(\mathcal{C})}{128} \right\rceil - len(\mathcal{C})$ and let $v = 128 \cdot \left\lceil \frac{len(\mathcal{A})}{128} \right\rceil - len(\mathcal{A})$
Define block $\mathcal{S} = \mathcal{GHASH}_{\mathrm{H}}(\mathcal{A} \| 0^v \| \mathcal{C} \| 0^u \| [len(\mathcal{A})]_{64} \| [len(\mathcal{C})]_{64})$
Let $\mathcal{T}' = \mathcal{MSB}_t(\mathcal{GCTR}_{\mathrm{K}}(\mathcal{J}_0, \mathcal{S}))$
if $\mathcal{T} = \mathcal{T}'$ **then**
    return $\mathcal{M}$;
else
    return FAIL;
end

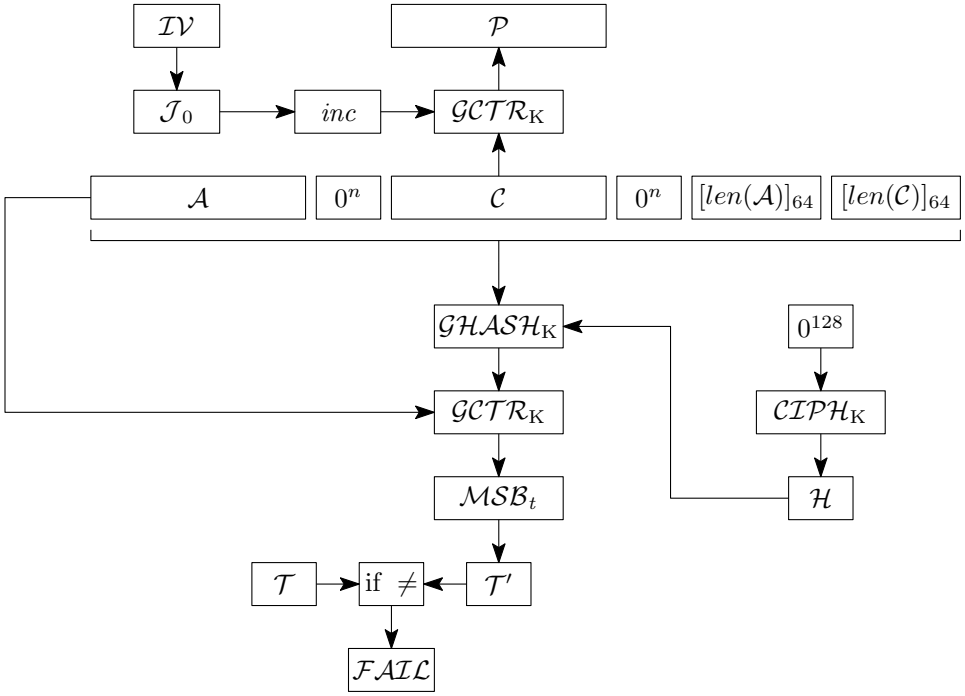

Figure 3.4: AES-GCM-AD$_{\mathcal{K}}(\mathcal{IV}, \mathcal{C}, \mathcal{A}, \mathcal{T}) = \mathcal{P}$ or FAIL

Chapter 4

Proposed Scheme

As mentioned before, the AES Galois Counter Mode (GCM) is one of the first AE scheme which has received a great attention in the literature nowadays. Although the approach is a good scheme in terms of combining the decryption and integrity verification in a single step but it has some drawbacks.This approach is based on checking the integrity of the packet at the end after processing the whole packet. For example, If the packet size is n (n being very large) bytes then it will first process whole n bytes calculating the MAC tag on cipher-text and then check for the integrity of the packet. To counteract this drawback, one may use the proposed generic scheme a.k.a fail-fail authenticated encryption($\mathcal{FFAE}$) scheme which provides a fail-fast mechanism to discard the packet at an early stage if packet is malformed. The approach is based on generating two Tags (T_1) and (T_2) for MAC verification. If (T_1) is correct, then only the whole packet is processed to verify the integrity of the packet.Thus, speeding-up the MAC verification process in AES-GCM algorithm. Using the proposed scheme in this chapter, one can easily discard the packet at an early stage rather than processing the whole packet and then making a decision on whether to accept/reject the packet based on the MAC tag.

4.1 Methodology

We used two extra operations to be performed along with the current AES-GCM algorithm as shown in Figure 4.1.

4.1.1 Notations used

Table 4.1 summarizes the notation used in this thesis document.

Table 4.1: Notations used in proposed scheme

Symbol	Meaning
$\mathcal{N}$ or $\mathcal{IV}$	Unique nonce/initialization vector of length n=128 bits
$\mathcal{K}$	Distinct Key of length k bits
$\mathcal{A}$	Additional authenticated data of length between 0 to 2^{64}
$\mathcal{M}$	Plaintext message, can have any number of bits between 0 and $2^{39} - 256$
$\mathcal{H}_{\mathcal{K}}$	A keyed hash function
$\mathcal{N}^{'}$	Result obtained after performing H_K(N or IV) of length $n^{'}$
$\mathcal{N}^{''}$	Result obtained after performing $\mathrm{D}_K(\mathrm{T}_1)$ of length $n^{''}$ for verification
$\mathcal{E}_{\mathcal{K}}$	A symmetric key encryption function
$\mathcal{D}_{\mathcal{K}}$	A symmetric key decryption function
$\mathcal{T}_1$	An authentication tag of length t_1=128 bits, obtained after $\mathrm{E}_K(\mathrm{N}^{'})$
$\mathcal{AE}_{\mathcal{K}}$	Authenticated Encryption function
$\mathcal{C}$	Ciphertext, length is same as of Plaintext
$\mathcal{T}_2$	An authentication tag of length t_2 which varies from 64 to 128 bits

4.1.2 Design specifications

This section gives a detailed explanation of the scheme proposed and its layout overview.

4.1.2.1 Parameters and Components

A particular instance of this scheme consists of deciding following algorithms:

- A keyed hash function ($\mathcal{H}_{\mathcal{K}}$)
- An encryption function($\mathcal{E}_{\mathcal{K}}$)
- An authenticated encryption algorithm ($\mathcal{AE}_{\mathcal{K}}$)

Sender Side ciphertext and tag generation

Scheme demands following parameters to be taken as input on sender side :

- a key $\mathcal{K}$
- a nonce $\mathcal{N}$ of 128 bits
- a datagram($\mathcal{A}$, $\mathcal{M}$) where
  - $\mathcal{A}$ is an additional authenticated data
  - $\mathcal{M}$ is the plaintext message

Produces following output:

- intermediate output $\mathcal{N}^{'}$
- final output($\mathcal{T}_1$, $\mathcal{C}$, $\mathcal{T}_2$) where
  - $\mathcal{T}_1$ is initial authentication tag calculated on $\mathcal{N}$
  - $\mathcal{C}$ is the ciphertext(or encrypted payload)
  - $\mathcal{T}_2$ is an authentication tag

In summary, it can be represented by following equations:

$$\mathcal{H} : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^{n'}$$

with

$$\mathcal{H}(\mathcal{K}, \mathcal{N}) = (\mathcal{N}')$$

$$\mathcal{E} : \{0,1\}^k \times \{0,1\}^{n'} \rightarrow \{0,1\}^{t_1}$$

with

$$\mathcal{E}(\mathcal{K}, \mathcal{N}') = (\mathcal{T}_1)$$

$$\mathcal{AE} : \{0,1\}^k \times \{0,1\}^{n'} \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^{t_2}$$

with

$$\mathcal{AE}(\mathcal{K}, \mathcal{N}', \mathcal{A}, \mathcal{M}) = (\mathcal{C}, \mathcal{T}_2)$$

Receiver Side tag verification and plaintext generation

Following parameters are taken as input :

- a key $\mathcal{K}$
- a nonce $\mathcal{N}$ of 128 bits
- a datagram($\mathcal{A}$, $\mathcal{C}$, $\mathcal{T}_1$, $\mathcal{T}_2$) where
  - $\mathcal{A}$ is an additional authenticated data

- $\mathcal{C}$ is the ciphertext(or encrypted payload)
- $\mathcal{T}_1$ is initial verification authentication tag
- $\mathcal{T}_2$ is an authentication tag

Produces following output:

- intermediate output $\mathcal{N}^{'}$
- intermediate output $\mathcal{N}^{''}$
- final output($\mathcal{M} \parallel \perp$) where
  - $\mathcal{M}$ is the decrypted plaintext message, tag verification is successful
  - $\perp$, failure if tag verification is failed

In summary, this can be represented by following equations:

$$\mathcal{H} : \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^{n'}$$

with

$$\mathcal{H}(\mathcal{K}, \mathcal{N}) = (\mathcal{N}')$$

$$\mathcal{D} : \{0,1\}^k \times \{0,1\}^{t_1} \rightarrow \{0,1\}^{n''}$$

with

$$\mathcal{D}(\mathcal{K}, \mathcal{T}_1) = (\mathcal{N}^{"})$$

if $\mathcal{N}' = \mathcal{N}''$

$$\mathcal{AE} : \{0,1\}^k \times \{0,1\}^{n'} \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^{t_2}$$

with

$$\mathcal{AE}(\mathcal{K}, \mathcal{N}', \mathcal{A}, \mathcal{C}) = \begin{cases} M, & \text{if } \mathcal{T}_2 \text{ is correct} \\ \perp, & \text{otherwise} \end{cases}$$

$\perp$, otherwise

4.1.2.2 Design overview

Refer Figure 4.1 for a brief description of transformations done on the message and IV at sender side as explained below :
Firstly, hash function $\mathcal{H}_\text{k}$ is applied on $\mathcal{N}$ or $\mathcal{IV}$ in order to generate $\mathcal{N}^{'}$, which is further used as initialization vector for $\mathcal{AE}$. Then, $\mathcal{N}^{'}$ generated is encrypted using symmetric key encryption algorithm $\mathcal{E}_\text{k}$ chosen to generate

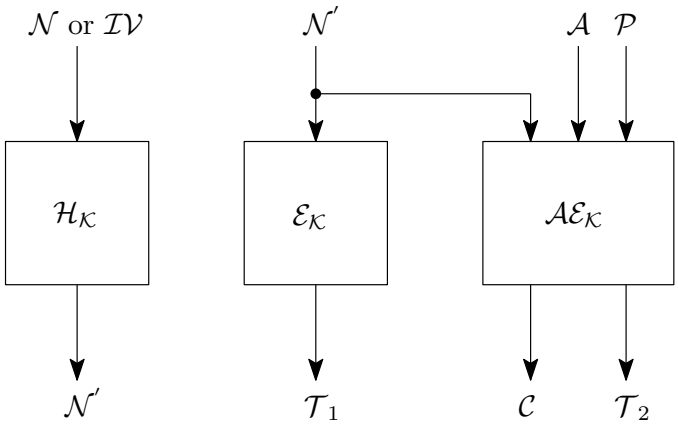

Figure 4.1: Proposed Scheme

the first tag $\mathcal{T}_1$, used to discard garbage packet at an early stage. After the initial tag is generated, encrypt the plaintext message $\mathcal{M}$ to get the ciphertext $\mathcal{C}$ and its associated authentication tag $\mathcal{T}_2$.

After performing all the transformations, $\mathcal{T}_1 \parallel \mathcal{C} \parallel \mathcal{T}_2$ is send to the receiver.

At receiver side, following transformations are applied on the packet $\mathcal{T}_1 \parallel \mathcal{C} \parallel \mathcal{T}_2$ in order to verify the integrity of the message and decrypt ciphertext to generate plaintext. Refer Figure 4.2.
Firstly, hash function $\mathcal{H}_\mathrm{k}$ is applied on $\mathcal{N}$ or $\mathcal{IV}$ in order to generate $\mathcal{N}^{'}$, which is further used for verification purpose before applying $\mathcal{AE}$. Next, $\mathcal{T}_1$ received in the packet is decrypted using decryption algorithm $\mathcal{D}_\mathrm{k}$ to generate $\mathcal{N}^{''}$. Then, $\mathcal{N}^{'}$ generated earlier is compared with $\mathcal{N}^{''}$.If the two exactly matches, then only $\mathcal{AE}$ is applied, otherwise the packet is discarded right away. $\mathcal{N}^{'}$ generation and $\mathcal{T}_1$ decryption can be done in parallel. In case of $\mathcal{AE}$, $\mathrm{N}^{''}$ generated earlier is used as the initialization vector for the decryption of ciphertext $\mathcal{C}$ to produce plaintext message $\mathcal{M}$. Also $\mathcal{AE}$ verifies the correctness of authentication tag $\mathcal{T}_2$. If it is correct, then message is returned otherwise failure is returned.

4.1.2.3 High level structure of proposed design

Step-by-step procedure followed for ciphertext and tag generation at server side is specified in function CiphTagGen. Also, for tag verification and

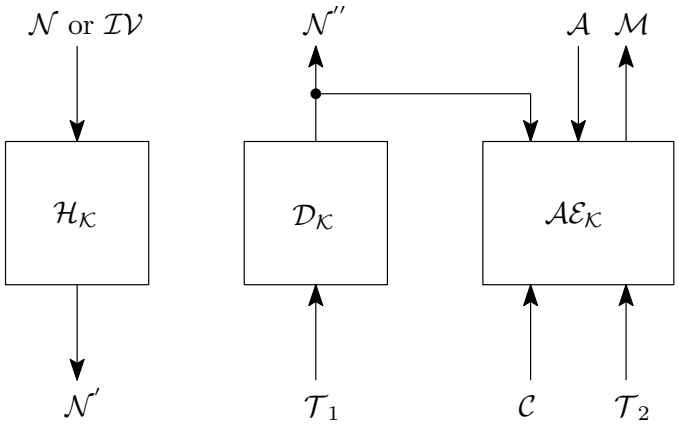

Figure 4.2: Proposed Scheme for verification

plaintext generation at receiver side is depicted in CiphDecTagVerify function shown as below:

4.2 Security Analysis

In this section, we have provided an intuitive proof of the proposed scheme.

4.2.1 Privacy of FFAE

Theorem 4.2.1 *Let $\mathcal{FFAE}$ be an authenticated encryption scheme, which consists of cryptographically secure primitives keyed hash function $\mathcal{H}$, authenticated encryption scheme $\mathcal{AE}$ and a symmetric encryption scheme $\mathcal{E}$. Let $\mathcal{A}$ be a probabilistic polynomial time adversary which tries to break the privacy of $\mathcal{FFAE}$. The advantage of adversary $\mathcal{A}$ in defeating $\mathcal{FFAE}$ is given by :*

$$\mathsf{Adv}^{priv}_{FFAE}(\mathcal{A}) \quad \leq \quad \mathsf{Adv}^{priv}_{AE}(\mathcal{A}) + \mathsf{Adv}^{prf}_{H_K}(\mathcal{A}) + \frac{q^2}{2^{n'}} \tag{4.1}$$

where $\mathsf{Adv}^{priv}_{AE}(\mathcal{A})$ be the privacy advantage $\mathcal{A}$ against $\mathcal{AE}$, $\mathsf{Adv}^{prf}_{H_K}(\mathcal{A})$ be the advantage of $\mathcal{A}$ in distinguishing $\mathcal{H}_\mathcal{K}$ from PRF, and q is the total number of queries made to $\mathcal{FFAE}$ and n' is the size of the output of keyed hash function.

Proof (sketch). Let FFAE be an authenticated encryption scheme. The privacy advantage of $\mathcal{A}$ against FFAE depends on following two cases:

Algorithm 17: Ciphertext and Tag generation algorithm

1 CiphTagGen $(\mathcal{N}, \mathcal{M}, \mathcal{A})$;
Prerequisites: approved keyed hash function $\mathcal{H}$;
approved symmetric encryption function $\mathcal{E}$;
approved authenticated encryption algorithm $\mathcal{AE}$;
key $\mathcal{K}$
Input : nonce $\mathcal{N}$;
plaintext message $\mathcal{M}$;
additional authenticated data $\mathcal{A}$
Output : ciphertext $\mathcal{C}$;
initial authentication tag $\mathcal{T}_1$;
authentication tag on C $\mathcal{T}_2$
2 $\mathcal{N}' \leftarrow \mathcal{H}_{\mathrm{k}}(\mathcal{N})$
3 $\mathcal{T}_1 \leftarrow \mathcal{E}_{\mathrm{k}}(\mathcal{N}')$
4 $\mathcal{C}, \mathcal{T}_2 \leftarrow \mathcal{AE}_{\mathrm{k}}(\mathcal{N}', \mathcal{A}, \mathcal{M})$
5 return $\mathcal{T}_1, \mathcal{C}, \mathcal{T}_2$

- $\mathcal{FFAE}$ **is not nonce-respecting** : In this case, if nonce is non-respecting, i.e., nonce N is never repeated. Therefore, the security bound for privacy of FFAE depends only on the privacy bound of authenticated encryption scheme AE. Formally, the advantage of the adversary $\mathcal{A}$ is given by

$$\mathsf{Adv}^{priv}_{FFAE}(\mathcal{A}) \quad \leq \quad \mathsf{Adv}^{priv}_{AE}(\mathcal{A}) \tag{4.2}$$

- $\mathcal{FFAE}$ **is nonce-respecting** : In this the privacy bound will depend on PRF advantage and the collision bound on the tag generated of keyed hash function. Formally, the advantage of the adversary $\mathcal{A}$ is given by the

$$\mathsf{Adv}^{priv}_{FFAE}(\mathcal{A}) \quad \leq \quad \mathsf{Adv}^{prf}_{H_K}(\mathcal{A}) + \frac{q^2}{2^{n'}} \tag{4.3}$$

4.2.2 Authenticity of FFAE

Theorem 4.2.2 *Let $\mathcal{FFAE}$ be an authenticated encryption scheme, which consists of cryptographically secure primitives keyed hash function $\mathcal{H}$, authenticated encryption scheme $\mathcal{AE}$ and a symmetric encryption scheme $\mathcal{E}$.*

Algorithm 18: Ciphertext decryption and Tag verification

CiphDecTagVerify $(\mathcal{N}, \mathcal{C}, \mathcal{T}_1, \mathcal{A}, \mathcal{T}_2)$;
Prerequisites: approved keyed hash function $\mathcal{H}$;
approved symmetric encryption function $\mathcal{E}$;
approved authenticated encryption algorithm $\mathcal{AE}$;
key $\mathcal{K}$
Input : nonce $\mathcal{N}$;
ciphertext $\mathcal{C}$;
initial authenticated tag $\mathcal{T}_1$;
additional authenticated data $\mathcal{A}$;
authentication tag $\mathcal{T}_2$
Output : plaintext message $\mathcal{M}$ or a Failure message
$\mathcal{N}' \leftarrow \mathcal{H}_{\mathrm{k}}(\mathcal{N})$
$\mathcal{N}'' \leftarrow \mathcal{D}_{\mathrm{k}}(\mathcal{T}_1)$
if $\mathcal{N}' = \mathcal{N}''$ **then**
  $\mathcal{M}, \mathcal{T}'_2 \leftarrow \mathcal{AE}_{\mathrm{k}}(\mathcal{N}'', \mathcal{A}, \mathcal{C})$
  if $\mathcal{T}_2 = \mathcal{T}'_2$ **then**
    return $\mathcal{M}$
  else
    return FAILURE;
  end
else
  return FAILURE;
end

Let $\mathcal{A}$ be a probabilistic polynomial time adversary which tries to break the authenticity of $\mathcal{FFAE}$. The advantage of adversary $\mathcal{A}$ in defeating $\mathcal{FFAE}$ is given by :

$$\mathsf{Adv}^{auth}_{FFAE}(\mathcal{A}) \quad \leq \quad \mathsf{Adv}^{auth}_{AE}(\mathcal{A}) + \mathsf{Adv}^{prf}_{H_K}(\mathcal{A}) + \frac{q_v}{2^{n'}} \tag{4.4}$$

where $\mathsf{Adv}^{prf}_{H_K}(\mathcal{A})$ be the advantage of $\mathcal{A}$ in distinguishing $\mathcal{H}_\mathcal{K}$ from PRF, $\mathsf{Adv}^{auth}_{AE}(\mathcal{A})$ be the authenticity advantage $\mathcal{A}$ against $\mathcal{AE}$, and q_v denote the number of verification queries made to $\mathcal{FFAE}$ out of total number of queries q made by $\mathcal{A}$, and n' is the size of the output of keyed hash function.

Proof (sketch). Let FFAE be an authenticated encryption scheme. The authenticity advantage of FFAE depends on following two cases:

- $\mathcal{FFAE}$ **is not nonce-respecting** : Without the loss of generality, if $\mathcal{N}$ is repeated, then $\mathcal{N}^{'}$ will also be repeated. Thus, the security bound for authenticity of FFAE depends upon the collision bound on the tag generated from authenticated encryption function AE. Formally, the advantage of the adversary $\mathcal{A}$ is given by the

$$\mathsf{Adv}^{auth}_{FFAE}(\mathcal{A}) \quad \leq \quad \mathsf{Adv}^{auth}_{AE}(\mathcal{A}) \tag{4.5}$$

- $\mathcal{FFAE}$ **is nonce-respecting** : In this case, we have another two cases, (i) $\mathcal{N}^{'}$ is repeated, and (ii) $\mathcal{N}^{'}$ is not repeated. In any case, the authenticity for FFAE will depend only on the PRF advantage and the collision bound on the tag generated of keyed hash function. Let q_v denote the number of verification queries made to the system. Formally, the advantage of the adversary $\mathcal{A}$ is given by the

$$\mathsf{Adv}^{auth}_{FFAE}(\mathcal{A}) \quad \leq \quad \mathsf{Adv}^{prf}_{H_K}(\mathcal{A}) + \frac{q_v}{2^{n'}} \tag{4.6}$$

Chapter 5

Results and Performance Analysis

In this section we describe the results of our approach and also discuss about the performance analysis of our work.

5.1 Target Devices

The scheme was implemented in the following devices:

1. **Intel Core i3-380M**: 2.53 GHz, 4 GB RAM, MSVC Compiler
2. **Raspberry Pi 2, BCM2836**: 1 GHz, 1 GB RAM, gcc (ARM)
3. **AVR32, AVR32UC3A0512**: 12 MHz, 64 Kb RAM avr32-gcc
4. **ATXMEGA128A1**: 32 MHz, 4 Kb RAM, avr-gcc
5. **ATMEGA328P**: 16 MHz, 2 Kb RAM avr-gcc

5.2 Results Analysis

We calculated the performance factor of a particular instance of cipher in our proposed scheme i.e. we used HMAC-SHA256 as hash function, AES as encryption algorithm and AES-GCM as authenticated encryption function. The reference implementation used OpenSSL and AVRCryptoLib (used to get the optimized and high performance code for well known cryptographic algorithms like AES and AES-GCM). We used /O2 optimization flags for compilation on all platforms.

Figure 5.1: Intel Core i3-380M@2.53 GHz

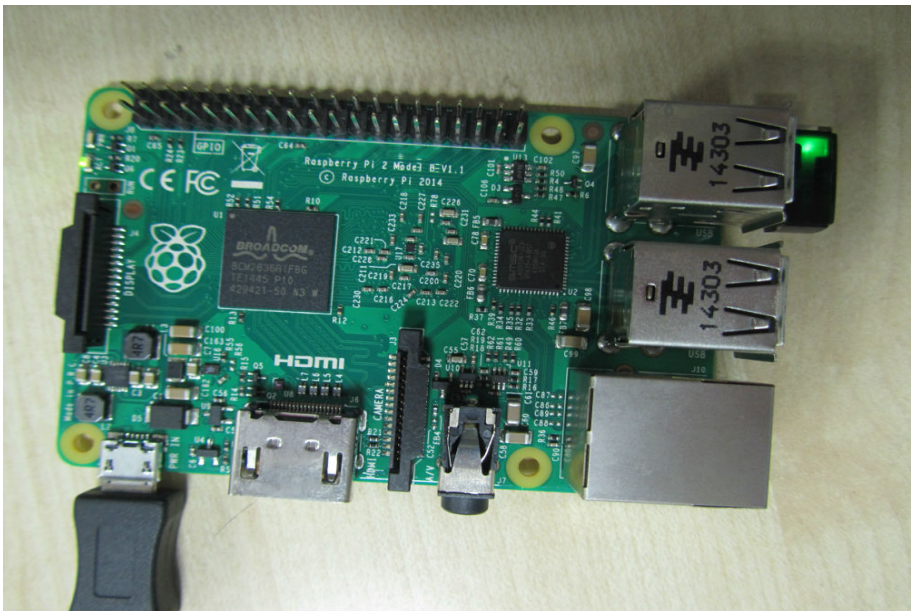

Figure 5.2: Raspberry Pi 2, BCM2836@1 GHz

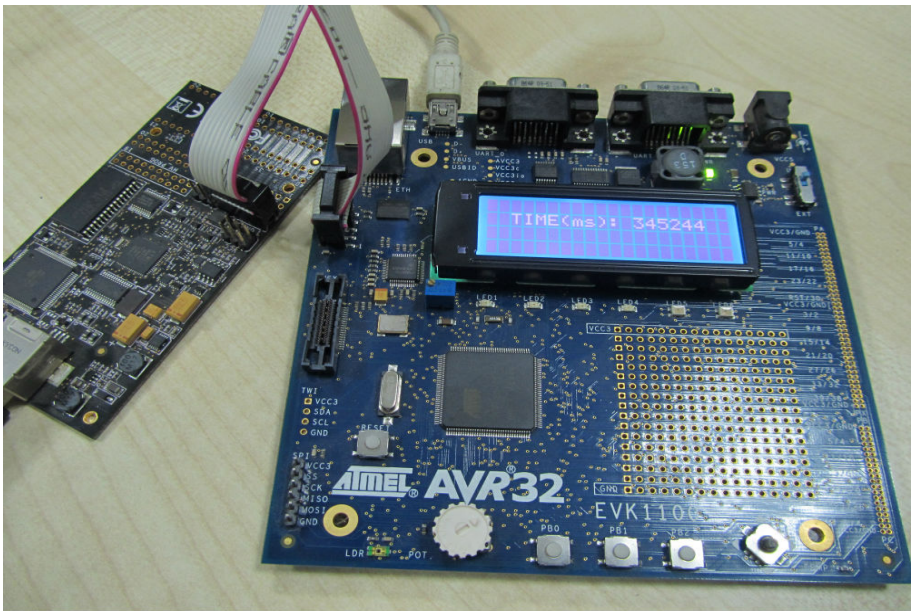

Figure 5.3: AVR32, AVR32UC3A0512@12 MHz

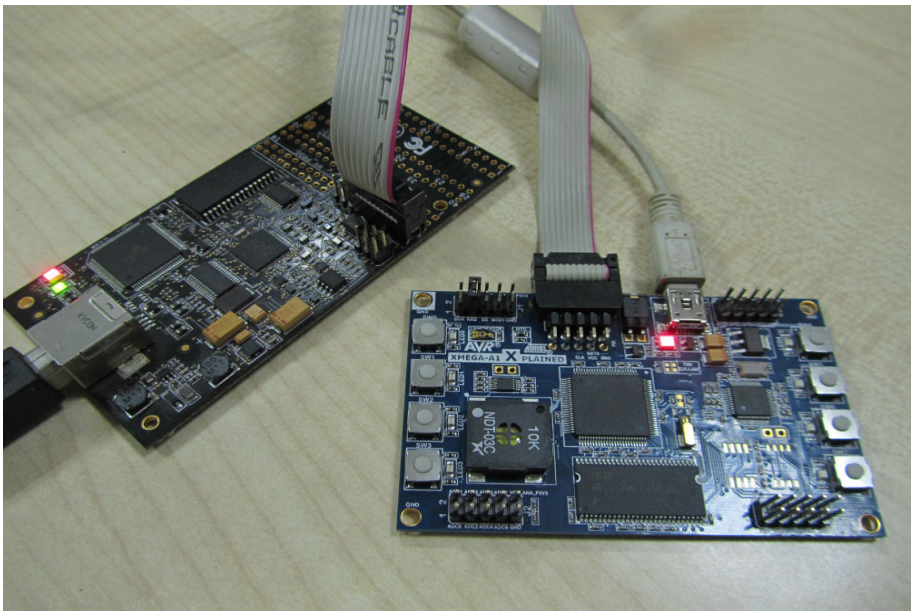

Figure 5.4: ATXMEGA128A1@32 MHz

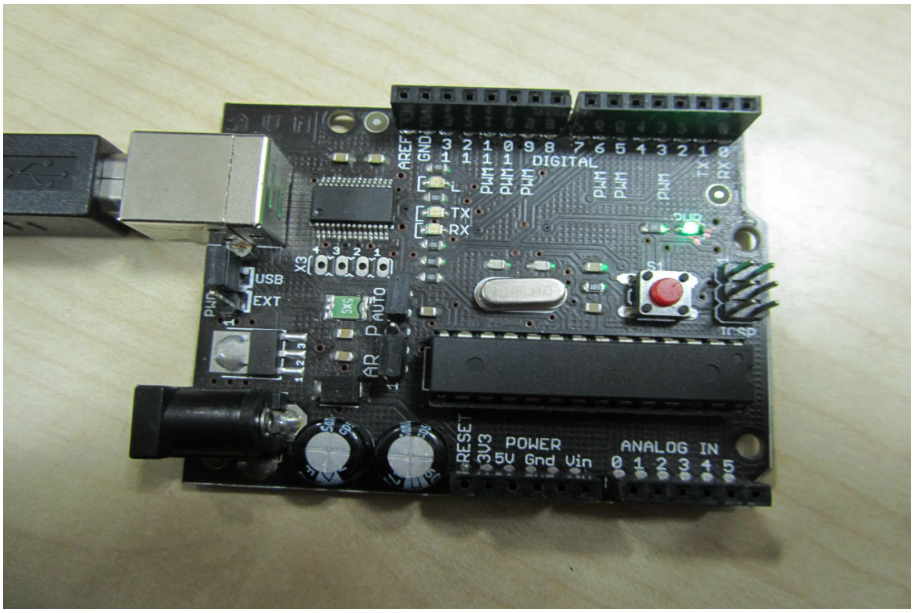

Figure 5.5: ATMEGA328P@16 MHz

Table 5.1 and 5.2 shows the comparison of Execution Time (in milliseconds) between our approach with fail-fast mechanism and currently used approach on various platforms for messages with varying length. We motivate using the proposed fail-fast mechanism to avoid the time consumed decryption and MAC verification process in case the packet is malformed.

Figure 5.6 and 5.7 presents the comparison between our approach of using AES-GCM with fail-fast mechanism and the way AES-GCM is currently used. From figure 5.7 it can be clearly seen that even if we introduce a little overhead in the packet size, then also the time taken for ciphertext and MAC tag generation is almost same for both the schemes for packet with larger size. Also, it can be seen from 5.6 that there is a significant deviation in execution time. Thus, depicting even with a small overhead we can achieve a huge performance improvement in case the packet is malformed and has a very large size. Therefore, proposed technique can be used to achieve fail-fast mechanism in any AE scheme being used these days.

Data Length[bytes]	16	32	64	128	512	1024	1536	4 KiB	8 KiB	16 KiB
XMEGA (ATXMEGA128A1) - 8bit @ 32 MHz										
Correct MAC										
AES-GCM	7.424	9.952	15.01	25.31	86.98	169.1	251.3	-	-	-
This work	15.07	17.66	22.82	33.02	94.98	176.9	258.9	-	-	-
Incorrect MAC										
AES-GCM	7.36	9.92	15.10	25.38	86.66	169.6	250.7	-	-	-
This work	7.744	7.712	7.712	7.744	7.712	7.712	7.712	-	-	-
ATMEGA328P - 8bit @ 16 MHz										
Correct MAC										
AES-GCM	15.55	20.93	31.62	53.38	-	-	-	-	-	-
This work	29.95	35.39	46.34	67.78	-	-	-	-	-	-
Incorrect MAC										
AES-GCM	15.42	20.8	31.87	53.57	-	-	-	-	-	-
This work	14.46	14.46	14.46	14.46	-	-	-	-	-	-
AVR32 (AVR32UC3A0512) - 32bit @ 12 MHz										
Correct MAC										
AES-GCM	1.978	2.253	2.801	3.899	10.49	19.20	-	71.71	141.7	167.5
This work	4.984	5.295	5.917	7.159	14.62	21.94	-	72.91	140.9	162.6
Incorrect MAC										
AES-GCM	1.949	2.224	2.772	3.869	10.46	19.17	-	71.68	141.7	167.5
This work	3.319	3.319	3.319	3.319	3.320	3.701	-	3.701	3.701	3.702

Table 5.1: Comparison of Execution Time in milliseconds

Data Length[bytes]	16	32	64	128	512	1024	1536	4 KiB	8 KiB	16 KiB	32 KiB	512 KiB	1 MiB
Raspberry Pi 2 - BCM2836 - @ 1GHz													
Correct MAC													
AES-GCM	0.445	-	0.436	0.438	0.56	0.655	0.763	1.379	-	-	7.969	120.305	239.341
This work	0.476	-	0.517	0.506	0.611	0.701	0.834	1.394	-	-	7.88	119.597	238.255
Incorrect MAC													
AES-GCM	0.414	-	0.428	0.436	0.585	0.652	0.763	1.382	-	-	7.85	120.26	239.337
This work	0.323	-	0.349	0.325	0.338	0.345	0.356	0.338	-	-	0.372	0.476	0.483
Intel Core i3 - M380 - @ 2.53GHz													
Correct MAC													
AES-GCM	0.384	-	0.382	0.398	0.435	0.435	0.451	0.489	-	-	2.039	35.62	67.28
This work	0.371	-	0.374	0.378	0.379	0.409	0.408	0.568	-	-	1.886	27.71	55.51
Incorrect MAC													
AES-GCM	0.227	-	0.244	0.273	0.334	0.327	0.349	0.48	-	-	2.071	36.64	64.23
This work	0.319	-	0.315	0.315	0.332	0.331	0.326	0.334	-	-	0.375	0.393	0.385

Table 5.2: Comparison of Execution Time in milliseconds

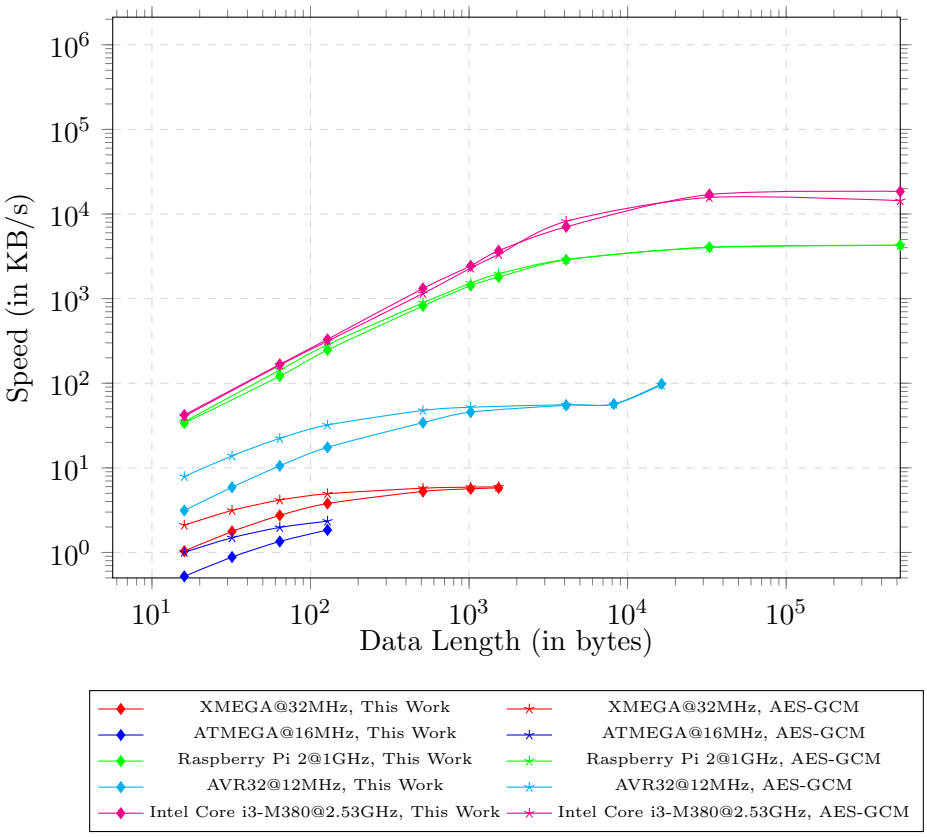

Figure 5.6: Performance Overhead of our approach with fail-fast mechanism v/s existing AE scheme

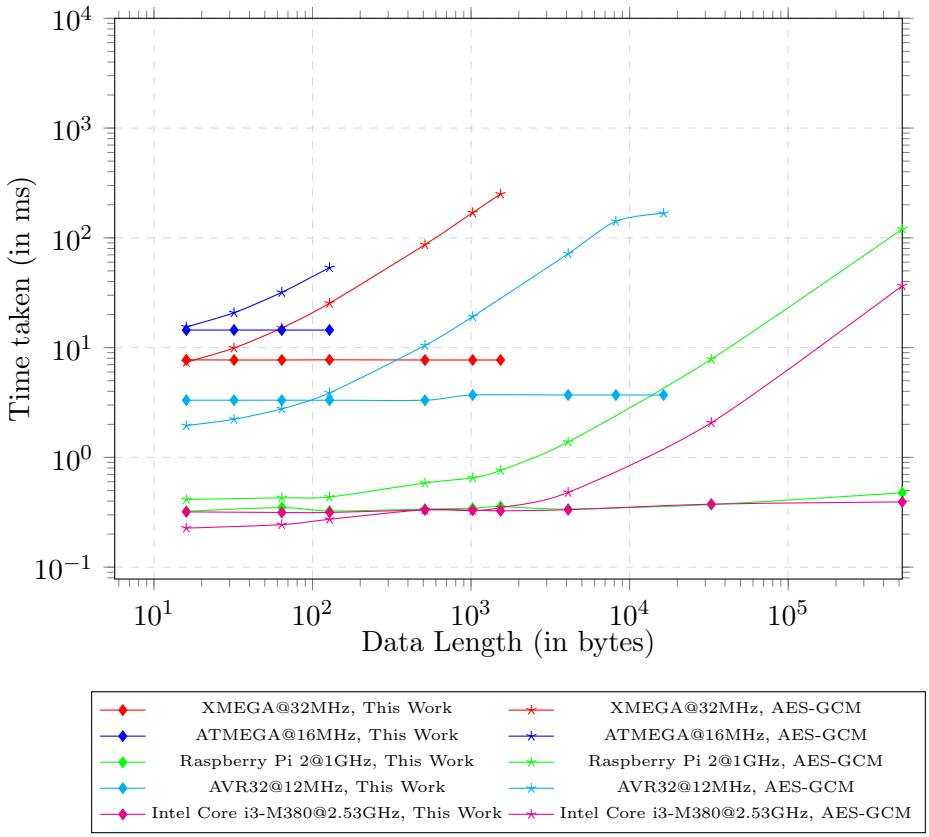

Figure 5.7: Execution Time of our approach with fail-fast mechanism v/s existing AE scheme

Chapter 6

Conclusion and Future Work

A new design approach for fail-fast mechanism in AE schemes has been introduced. It was shown in performance analysis that our approach outperforms the way AE schemes are used currently. Thus, if an attacker sends some garbage packet or if it is malformed, then using our approach it will be discarded at an early stage, thereby providing fail-fast mechanism to existing AE schemes. The results has been measured with a great accuracy.

In future, we will provide a formal proof of the security bounds introduced in section 4.2. Also, as an extension to this work, there can be multiple variants of the scheme, so we would analyse the variants and see which variant gives much better efficiency.

Bibliography

[1] John Black and Hector Urtubia. Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In *USENIX Security Symposium*, pages 327–338, 2002.

[2] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Advances in CryptologyCRYPTO'98*, pages 1–12. Springer, 1998.

[3] Brice Canvel, Alain Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a ssl/tls channel. In *Advances in Cryptology-Crypto 2003*, pages 583–599. Springer, 2003.

[4] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[5] Tim Dierks. The transport layer security (tls) protocol version 1.2. 2008.

[6] Thai Duong and Juliano Rizzo. Here come the ninjas. *Unpublished manuscript*, page 4, 2011.

[7] Morris Dworkin. Recommendation for block cipher modes of operation. methods and techniques. Technical report, DTIC Document, 2001.

[8] Burt Kaliski. Pkcs# 7: Cryptographic message syntax version 1.5. 1998.

[9] Tadayoshi Kohno, John Viega, and Doug Whiting. Cwc: A high-performance conventional authenticated encryption mode. In *Fast Software Encryption*, pages 408–426. Springer, 2004.

[10] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication. 1997.

[11] David McGrew and John Viega. The galois/counter mode of operation (gcm). *Submission to NIST. http://csrc. nist. gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec. pdf*, 2004.

[12] Kenneth G Paterson and Gaven J Watson. Immunising cbc mode against padding oracle attacks: A formal security treatment. In *Security and Cryptography for Networks*, pages 340–357. Springer, 2008.

[13] Serge Vaudenay. Security flaws induced by cbc paddingapplications to ssl, ipsec, wtls... In *Advances in CryptologyEUROCRYPT 2002*, pages 534–545. Springer, 2002.

[14] Tatu Ylonen and Chris Lonvick. The secure shell (ssh) transport layer protocol. 2006.