2018

Efficient Implementation of IEEE 802.11i Wi-Fi Security (WPA2-PSK) Standard Using FPGA

Atal Bajracharya
Grand Valley State University

Efficient Implementation of IEEE 802.11i Wi-Fi Security (WPA2-PSK) Standard Using FPGA

Atal Bilas Bajracharya

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Master of Science in Engineering

School of Engineering

December 2018

Acknowledgements

The successful completion of this thesis required guidance and assistance from a lot people and I am extremely privileged to have got this throughout the course of my thesis. All that I have done is only possible due to such supervision and assistance.

I would like to express my sincere appreciation to my thesis advisor Dr. Chirag Parikh for his wealth of knowledge, immense support and constant guidance without which I would not have been able to complete my thesis. I would also like to thank the rest of my thesis committee members, Dr. Nabeeh Kandalaft and Dr. Christian Trefftz for their invaluable comments and feedback which helped me correctly cover my subject matter. I would also like to express my immense gratitude towards my graduate advisor Dr. Shabbir Choudhuri, for guiding me throughout my master's degree program at Grand Valley State University. Finally, I would like to thank my family and friends for their constant love and support throughout my life.

Abstract

The rationale behind the thesis was to design efficient implementations of cryptography algorithms used for Wi-Fi Security as per IEEE 802.11i Wi-Fi Security (WPA2-PSK) standard. The focus was on software implementation of Password-Based Key Derivation Function 2 (PBKDF2) using Keyed-Hash Message Authentication Code (HMAC)-SHA1, which is used for authentication, and, hardware implementation of AES-256 cipher, which is used for data confidentiality.

In this thesis, PBKDF2 based on HMAC-SHA1 was implemented on software using C programming language, and, AES-256 was implemented on hardware using Verilog HDL. The overall implementation was designed and tested on Nexys4 FPGA board. The performance of the implementation was compared with other existing designs. Latency (us) was used as the performance metric for PBKDF2, whereas, throughput (Gb/s), resource utilization (Number of Slices), efficiency (Kb/s per slice) and latency (ns) were used as performance metrics for AES-256. MRF24WG0MA PMOD Wi-Fi module was the 2.4 GHz Wi-Fi module which was interfaced with Nexys4 FPGA board for wireless communication.

When the correct security credentials were entered in the implemented system interfaced to the Wi-Fi module, it was successfully authenticated by a 2.4 GHz wireless router (or mobile hotspot) configured to work in WPA2-PSK security mode. Once this system was authenticated, the implemented AES-256 cipher within the system was used to provide a layer of encryption over the data being communicated in the network.

Table of Contents

List of Figures

List of Tables

Abbreviations

ABBREVIATION	EXPLANATION
AES	Advanced Encryption System
ASCII	American Standard Code for Information Interchange
BRAM	Block RAM
FF	Flip Flop
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HMAC	Keyed-Hash Message Authentication Code
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
LAN	Local Area Network
LUT	Look Up Table
MIC	Message Integrity Check
PBKDF2	Password-Based Key Derivation Function 2
PMK	Pairwise Master Key
PMOD	Peripheral Module
PRF	Pseudo Random Function
PSK	Pre-Shared Key
PTK	Pairwise Transient Key
RISC	Reduced-instruction-set Computing
SDK	Software Development Kit
SHA1	Secure Hashing Algorithm 1
SSID	Service Set Identification
TCP	Transmission Control Protocol
WPA2	Wi-Fi Protected Access 2

Chapter 1: Introduction

1.1 Background

As most technologies have continued to transition from traditional wired systems to wireless ones, the number of wireless devices has grown by leaps and bounds over the last decade. Wireless devices have become a part of our day-to-day lives with its presence seen in household, educational and business institutions, to name a few. These devices are inter-connected with one another and share a variety of data, ranging from mundane to very personal and confidential information. Such interconnected devices that share data among themselves form a network. There can be various types of networks based on topology, size, area, organization, etc. One such type of network based on area is called Local Area Network (LAN). Such network is confined within a localized area such as a room, building or a group of buildings. However, it can be inter-connected to other LANs using wired or wireless media. If wireless medium is used to connect such LANs, then the overall network is called Wireless LAN (WLAN) [1].

The communication between the devices within a network is governed by a set of rules called communication protocols. The devices within a network must adhere to such protocols to successfully share and interpret data among other devices connected to the network. To maintain interoperability between the devices manufactured by various vendors, standardized communication protocols are defined for different type of networks. One such protocol for communication between wireless devices over LAN is the IEEE 802.11 protocol and is commonly known as ***Wi-Fi*** [2]. An example of ***Wi-Fi*** network is shown in Figure 1.

Figure 1: Example of Wi-Fi Network

1.2 Wireless Security Principle

Security is paramount in any type of network, but it is more so in the case of wireless networks, as they are far more vulnerable to attack in comparison to wired networks. In a wired network, the communicating devices must be physically connected using a cable. Hence, it is easier to verify the identity of the device to which the data is being communicated, as opposed to in wireless networks, where this is not quite easy. Also, unlike in wired networks, where the data is communicated through copper wires or optical fibers, in wireless networks, the wireless devices use RF signals in open air as their communication medium. So, theoretically any transceiver which is within the range of this RF signal and tuned to its frequency can read and/or meddle with the data being communicated [3]. Hence, for a secure communication, it is necessary to identity whether a device trying to connect to the network has proper security

credential or not. This process is called authentication [3]. After a wireless device is authenticated to a network, the data being communicated within that network must be made confidential using a secure cryptography algorithm [3].

1.3 WPA2-PSK Overview

The current standardized security protocol for ***Wi-Fi*** is IEEE 802.11i standard. This is also commonly known as ***Wi-Fi*** Protected Access II (WPA2). WPA2 was launched in September 2004 and supports PSK technology and includes an advanced encryption mechanism using the Counter-Mode/CBC-MAC Protocol (CCMP) called the Advanced Encryption Standard (AES) [4]. The PSK technology (in personal networks) is used to verify the identity of the communicating wireless devices. In PSK, the authentication process is performed by the access point (wireless router, mobile hotspot, etc.). With PSK, we can configure the access point (wireless router or hotspot) with a passphrase of 8 to 63 printable ASCII characters [5]. Using a technology called ***PBKDF2***, that passphrase, along with the network SSID, is used to generate unique encryption keys for wireless clients. In ***WPA2-PSK*** security, the same set of SSID and PSK is shared between all ***Wi-Fi*** end devices and the access point as shown in Figure 2 [6]. The SSID is analogous to Username and PSK is analogous to Passphrase in Figure 2. The wireless devices are authenticated and granted access to the network, if the password to the particular SSID matches [5]. After authentication, AES cipher is used to maintain the confidentiality of the data being communicated within the network.

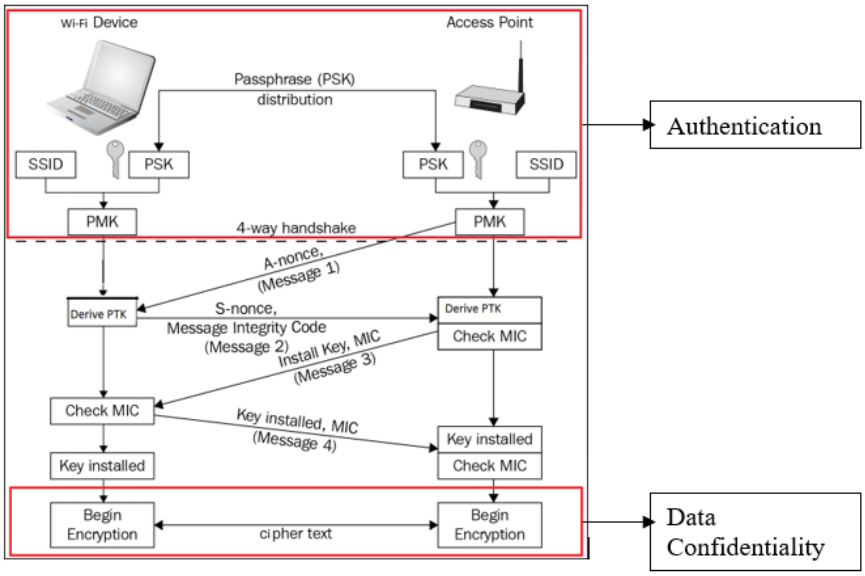


Figure 2: WPA2-PSK Security

In Figure 3, SSID and Passphrase goes through ***PBKDF2*** to derive the 256-bit PMK which is used as the main key for AES cipher. The validity of this key is confirmed using the 4-way handshake process (Figure 2) between the Wi-Fi device and the access point [6]. If the key matches, then, the ***Wi-Fi*** device is successfully authenticated by the access point.

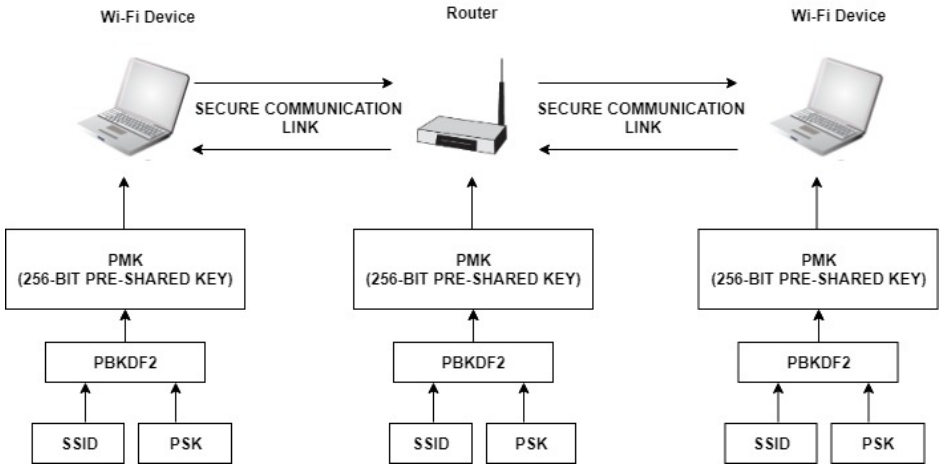


Figure 3: WPA2-PSK Authentication

After successful authentication, the data between ***Wi-Fi*** end devices and the access point is encrypted using AES cipher with the 256-bit PMK as the main key (Figure 4).

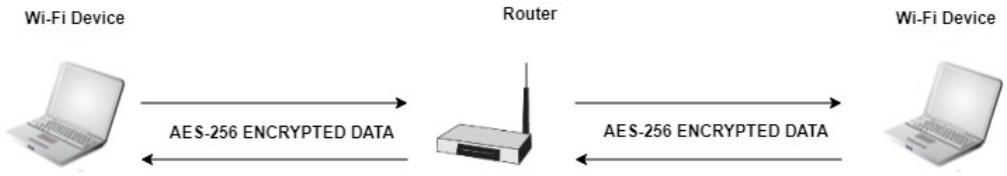


Figure 4: WPA2-PSK Data Confidentiality

1.4 Project Scope

The purpose of this thesis is to optimize the cryptography algorithms used in device authentication and data confidentiality in ***Wi-Fi*** networks configured with ***WPA2-PSK*** security. To achieve this, the main key derivation part of the authentication process, as well as, the AES cipher algorithm required for data confidentiality will be optimized. The scope of the implementation will encompass the following areas:

- Efficient software implementation of ***PBKDF2*** based on ***HMAC-SHA1*** which is used for device authentication.
- Efficient hardware implementation of ***AES-256*** cipher which is used for data confidentiality.

The performance of these implementations will be compared with other existing designs. Latency (us) will be used as the performance metric for ***PBKDF2***, whereas, throughput (Gb/s), resource utilization (Number of Slices), efficiency (Kb/s per slice) and latency (ns) will be used as performance metrics for ***AES-256***.

Chapter 2 describes the theory related to ***PBKDF2***, ***HMAC*** and ***SHA1*** used in ***WPA2-PSK*** device authentication. It also elaborates on ***AES-256*** key expansion, encryption and decryption.

Chapter 2: WPA2-PSK Theory

2.1 WPA2-PSK Device Authentication

Authentication is the process by which you prove that you are eligible to join a network (and that the network is legitimate) [3]. Pre-Shared Key (PSK) is a device authentication method used in ***WPA2-PSK*** networks, and it uses a passphrase of 8 to 63 printable ASCII characters to generate unique encryption keys [5]. The general idea of PSK mode is to use the same secret key on an access point and on a ***Wi-Fi*** device to authenticate the device and establish an encrypted connection for networking [6]. Hence, both ***Wi-Fi*** device and access point must prove to each other that they know the pre-shared key to ensure a secure connection. In ***WPA2-PSK***, the access point (wireless router, hotspot, etc.) with a network SSID is configured with a passphrase. Using ***PBKDF2***, that passphrase along with network SSID is used to generate the 256-bit Pairwise-Master-Key (PMK). The ***Wi-Fi*** device must also derive the same PMK using the same passphrase and SSID for the access point to authenticate the device.

PMKs are never transmitted across the network as the channel of communication is not secure before the authentication process has completed. Because, without authentication, sharing of PMK would be done through an unencrypted channel and susceptible to be discovered by outside parties [7]. To overcome this, ***WPA2-PSK*** uses 4-way handshake to verify whether the ***Wi-Fi*** device and the access point have the same PMK or not (Figure 2). The 4-way handshake is designed so that the access point and ***Wi-Fi*** device can independently prove to each other that they know the PMK, without ever disclosing it. In Figure 2, the 4-way handshake is broken down into 4 messages [7]:

- **Message 1 (From Access Point to Wi-Fi Device):** The first step is for the access point to generate a nonce value. The nonce value is a pseudo random value generated by a

publicly known and repeatable process. This pseudo random value is generated by the Pseudo Random Function 256 or PRF-256, as defined by WPA2 specifications. The nonce value generated by access point is called A-nonce. The access point sends a message containing this A-nonce value to the ***Wi-Fi*** device.

- **Message 2 (From Wi-Fi Device to Access Point):** The ***Wi-Fi*** device generates a nonce value using the same process as the access point and it is denoted as S-nonce. When the ***Wi-Fi*** device receives Message 1, it will generate Pairwise-Transient-Key (PTK). This key is required to be generated by both parties, and allows each party to verify that the other has the correct PMK. The creation of PTK is performed via another Pseudo Random Function (PRF), which uses a combination of the PMK, Access Point MAC Address, Wi-Fi Device MAC Address, A-nonce and S-nonce [8]. A part of the PTK is known as the message integrity check (MIC). This value, along with the S-Nonce is then transmitted back to the access point.
- **Message 3 (From Access Point to Wi-Fi Device):** When the access point receives Message 2, it has all the values required to generate the PTK. The access point then generates the PTK, and checks whether the MIC value in Message 2 matches the MIC value that it has just generated. If the two MIC values matches, this proves that the Wi-Fi device knows the value of the PMK. If the MIC value is correct, the access point, then sends Message 3 to the ***Wi-Fi*** device. Message 3 allows the ***Wi-Fi*** device to ensure that the access point is a trusted party. If the access point did not have a matching PMK, the MIC would be different. Message 3 also informs the ***Wi-Fi*** device that the communication channel is about to be encrypted.

- **Message 4 (From Wi-Fi Device to Access Point):** The final part of the handshake allows the ***Wi-Fi*** device to acknowledge that the access point is now going to use encryption for the communication. After the ***Wi-Fi*** device transmits Message 4, it will install the encryption keys on the channel. After the access point receives message 4, it will install the encryption keys as well. All further unicast communication is protected by this encryption, until the client disconnects from the access point [3].

2.1.1 PBKDF2 (Password-Based Key Derivation Function 2)

Using ***PBKDF2***, the passphrase and SSID are hashed 4096 times to produce a 256-bit PMK [9]. Internally, the ***PBKDF2*** key derivation function employed in ***WPA2-PSK*** utilizes 4096 iterations of ***HMAC-SHA1*** to obtain 160-bit hash outputs. Since the PMK in ***WPA2-PSK*** is of 256-bits, two rounds of ***PBKDF2*** are necessary [10]. Their outputs are concatenated, but for the second iteration the output is truncated to 96 bits to achieve the 256-bit PMK. The ***PBKDF2*** key derivation function is defined as follow:

D_K = PBKDF2(PRF, P, S, C, dk_{Len}) ..... (1)

where,

D_K: Derived key

PRF: Pseudorandom function of two parameters with output length h_{Len}

P: Password

S: Salt (sequence of bits)

C: Iteration count, a positive integer

dk_{Len}: Length of Derived key

To derive key from ***PBKDF2***, each h_{Len} bit block T_i of derived key D_K, is computed as follows:

$$D_K = T_1 \;||\; T_2 \;||\; \ldots \;||\; T_{dklen/hlen} \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (2)$$

$$T_i = F\ (P,\ S,\ C,\ i) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (3)$$

In equation (3), the function `F` is the Exclusive-OR operations of `C` iterations of `PRFs` (as shown in equation (4)). In the first iteration, the `PRF` uses `Password` as the key and `Salt` concatenated with `i` (encoded as a big-endian 32-bit integer) as the 2 parameters (as shown in equation (5)). For, subsequent iterations, `PRF` uses `Password` as the key and the output of the previous `PRF` computation as the salt (as shown in equations (6) and (7)). The block diagram for ***PBKDF2*** key derivation function is shown is Figure 5.

$$F\ (Password,\ Salt,\ C,\ i) = U_1 \oplus U_2 \oplus \ldots \oplus U_c \quad \ldots\ldots \quad (4)$$

where,

$$U_1 = PRF\ (Password,\ Salt \;||\; INT\_32\_BE(i)) \quad \ldots\ldots\ldots \quad (5)$$

$$U_2 = PRF\ (Password,\ U_1) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (6)$$

...

$$U_c = PRF\ (Password,\ U_c - 1) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots (7)$$

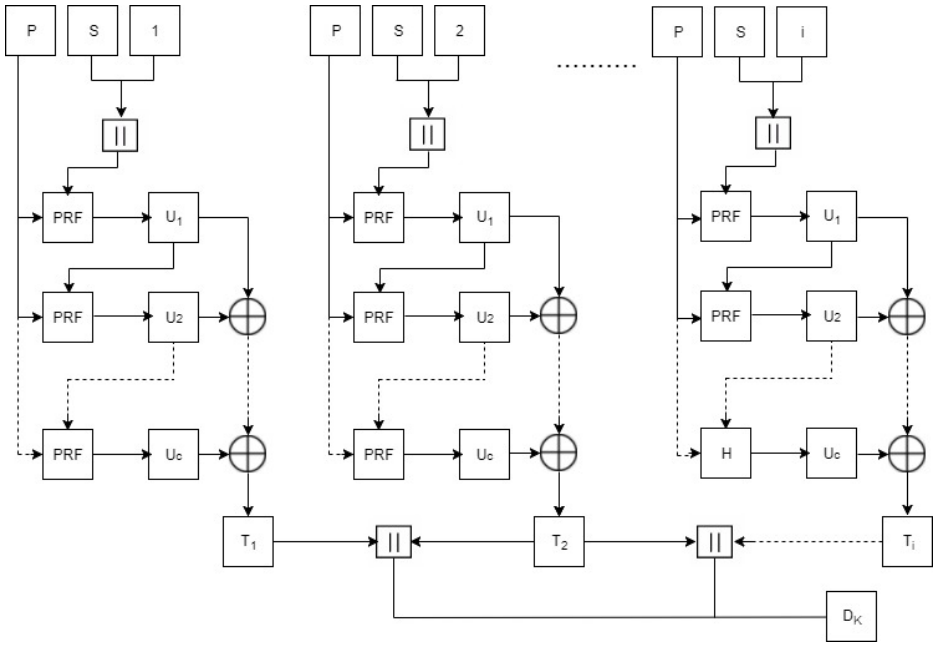


Figure 5: Block Diagram for PBKDF2

The overview of ***PBKDF2*** along with HMAC and SHA1 is shown in Figure 6 [11].

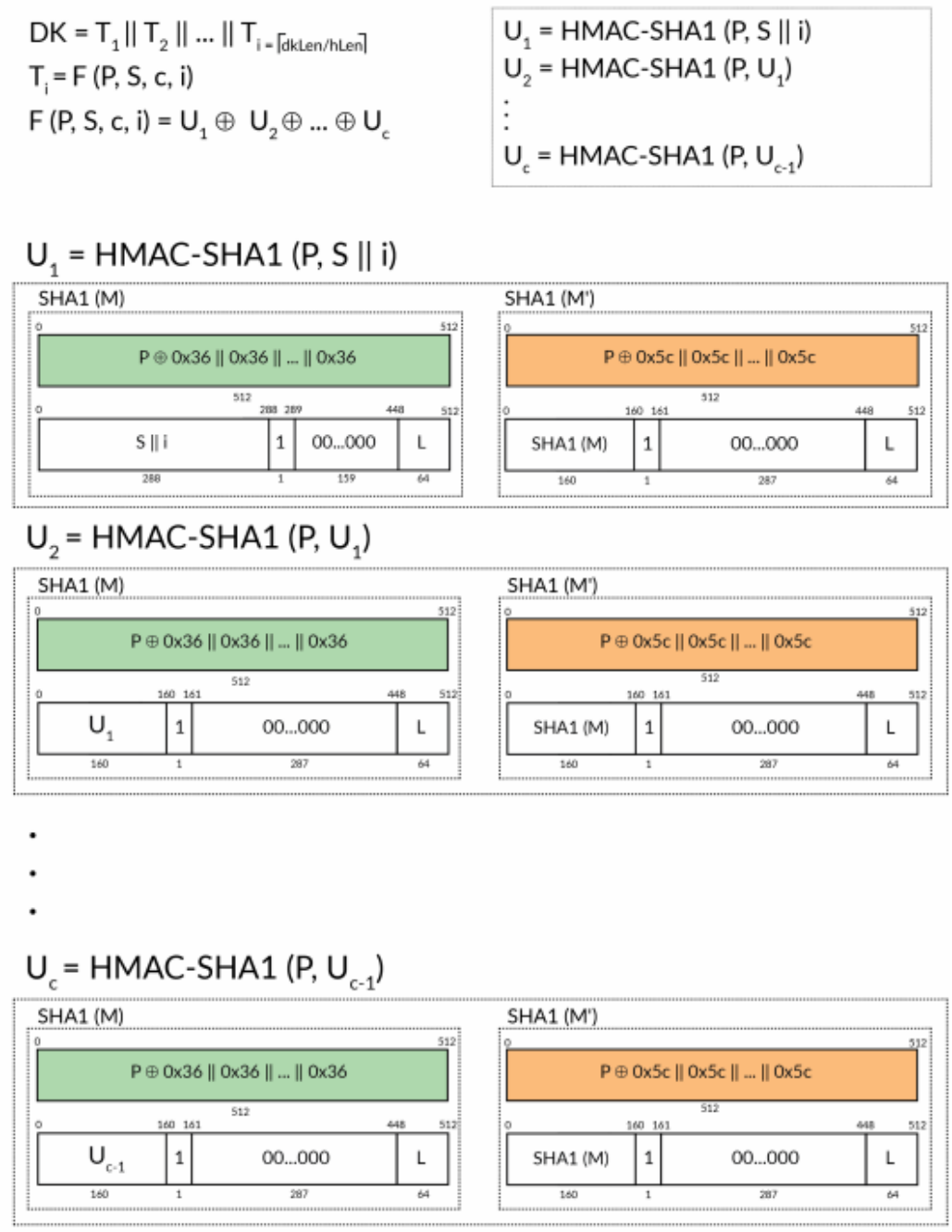


Figure 6: PBKDF2 with HMAC-SHA1

In case of ***WPA2-PSK***, the output and parameters in equation (1) are as follows:

```
PMK = PBKDF2(HMAC-SHA1, passphrase, ssid, 4096, 256) ..... (8)
```

2.1.2 HMAC (Keyed-Hashing for Message Authentication)

HMAC provides a mechanism to calculate a message authentication code (MAC) based around a cryptographic hashing function [7]. A message authentication code (MAC) is a short piece of information used to authenticate a message. MACs are used between two parties that share a secret key to validate information transferred between them [12]. The use of HMAC in PBKDF2 is shown in Figure 6. The definition of ***HMAC*** requires a cryptographic hash function denoted by `H`, the secret key denoted by `M` and the message to be authenticated denoted by `m`. The ***HMAC*** function is defined as follows:

```
HMAC (M, m) = H ((M' ⊕ opad) || H ((M' ⊕ ipad) || m)) ..........(9)

where,

H: a cryptographic hash function

M: the secret key

m: the message to be authenticated

M': another secret key, derived from the original key K
    (by padding K to the right with extra zeroes to the
    input block size of the hash function, or by hashing K if it
    is longer than that block size)

opad: the outer padding (0x5c5c5c…5c5c, one-block- long hexadecimal
constant

ipad: the inner padding (0x363636…3636, one-block- long hexadecimal
constant).
```

The block diagram for ***HMAC*** function is shown in Figure 7.

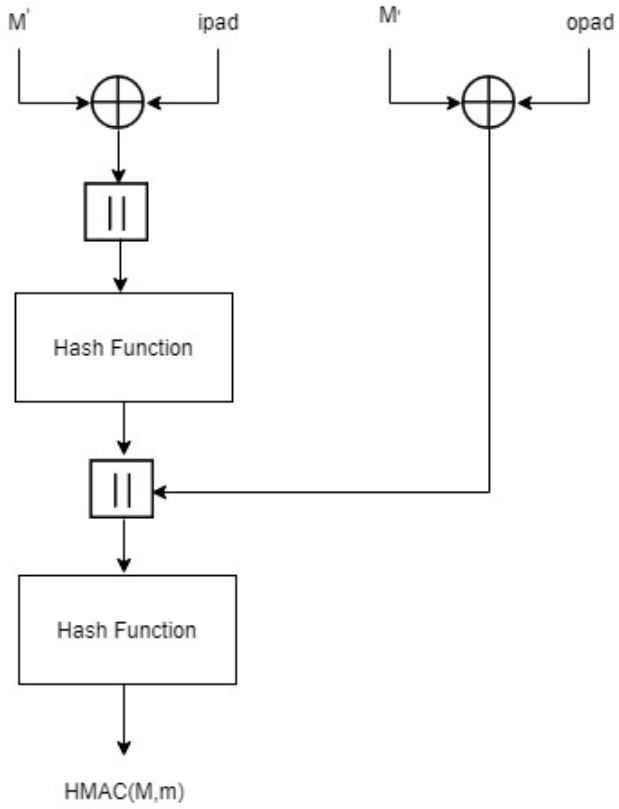


Figure 7: Block Diagram for HMAC

For ***WPA2-PSK***, the parameters in equation (9) are as follows:

$$\text{HMAC-SHA1 (passphrase, ssid)} = \text{SHA1 ((passphrase} \oplus \text{opad)} \;||\; \text{SHA1((passphrase} \oplus \text{opad} \oplus \text{ipad)} \;||\; \text{ssid)} \quad \ldots (10)$$

2.1.3 SHA1

Hashing algorithms are used to process a message and produce a condensed representation of the message which is called a message digest, and for a perfect hashing function, it should be only one-way and a unique digital signature of the message [7]. The use of ***SHA1*** in ***PBKDF2*** is shown in Figure 6. The ***SHA1*** hashing algorithm is valid for messages with a size less than 2^{64} bits, it operates on blocks of size 512 bits, it uses a word size of 32 bits, and has a resultant message digest of 160 bits [7]. ***SHA1*** algorithm primarily consists of 6 steps [13]:

Step1: Append Padding Bits: The original message is padded based on the following rules:

- The original message is first padded with one bit '1'.
- Zeros '0' are then padded to bring the length of message to 64 bits less than multiple of 512.

Step2: Append Length: A 64-bit value indicating the length of the original message is appended to end the message obtained from Step 1 based on the following rules:

- 64-bit value of the original message is appended at the end of the padded message. If overflow occurs, the lower order of the 64-bit value is appended.
- The lower 32-bit word of the 64-bit value is appended first followed by the upper 32-bit value.

Step3: Prepare Processing Functions: ***SHA1*** has 80 processing rounds. There are 4 mathematical operations assigned to each of the 4 sets of 20 rounds. These operations are as follows:

```
for 0 <= r <= 19,

F (r: B, C, D) = (B & C) | ((! B) & D) .............(11)

for 20 <= r <= 39,

F (r: B, C, D) = B ⊕ C ⊕ D.........................(12)

for 40 <= r <= 59,

F (r: B, C, D) = (B & C) | (B & D) | (C & D) .......(13)

for 60 <= r <= 79,

F (r: B, C, D) = B ⊕ C ⊕ D .......................(14)
```

Step4: Prepare Processing Constants: ***SHA1*** has 4 different constants assigned to 4 sets of 20 rounds. These constants are as follows:

```
for 0 <= r <= 19,

K(r) = 0x5A827999 ............................... (15)

for 20 <= r <= 39,

K(r) = 0x6ED9EBA1 ............................... (16)

for 40 <= r <= 59,

K(r) = 0x8F1BBCDC ............................... (17)

for, 60 <= r <= 79

K(r) = 0xCA62C1D6 ............................... (18)
```

Step5: Initialize Buffer: ***SHA1*** has five 32-bit buffers which are initialized as follows:

```
H0 = 0x67452301 ................................ (19)

H1 = 0xEFCDAB89 ................................ (20)

H2 = 0x98BADCFE ................................ (21)

H3 = 0x10325476 ................................ (22)

H4 = 0xC3D2E1F0 ................................ (23)
```

Step6: Process 512-bit block messages: The algorithm to process this 512-bit block of message is as follows:

```
For loop on k = 1 to N  /* 1st For loop */
    (W (0), W (1) ..., W (15)) = M[k] /*Divide M[k] into 16 words*/
    For t = 16 to 79 do: /* 2nd For loop */
      W(t) = (W(t-3) XOR W(t-8) XOR W(t-14) XOR W (t-16)) << 1
    End of For loop /* 2nd For loop */
    A = H0, B = H1, C = H2, D = H3, E = H4
    For t = 0 to 79 do: // 3rd for loop
         TEMP = A<<5 + f (t: B, C, D) + E + W(t) + K(t)
         E = D, D = C, C = B<<30, B = A, A = TEMP
    End of For loop // 3rd for loop
    H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D,
    H4 = H4 + E
End of for loop /* End of 1st For loop */
Output = H0 << 128 | H1 << 96 | H2 << 64 | H3 << 32 | H4
```

The block diagram for ***SHA1*** processing function is given in Figure 8.

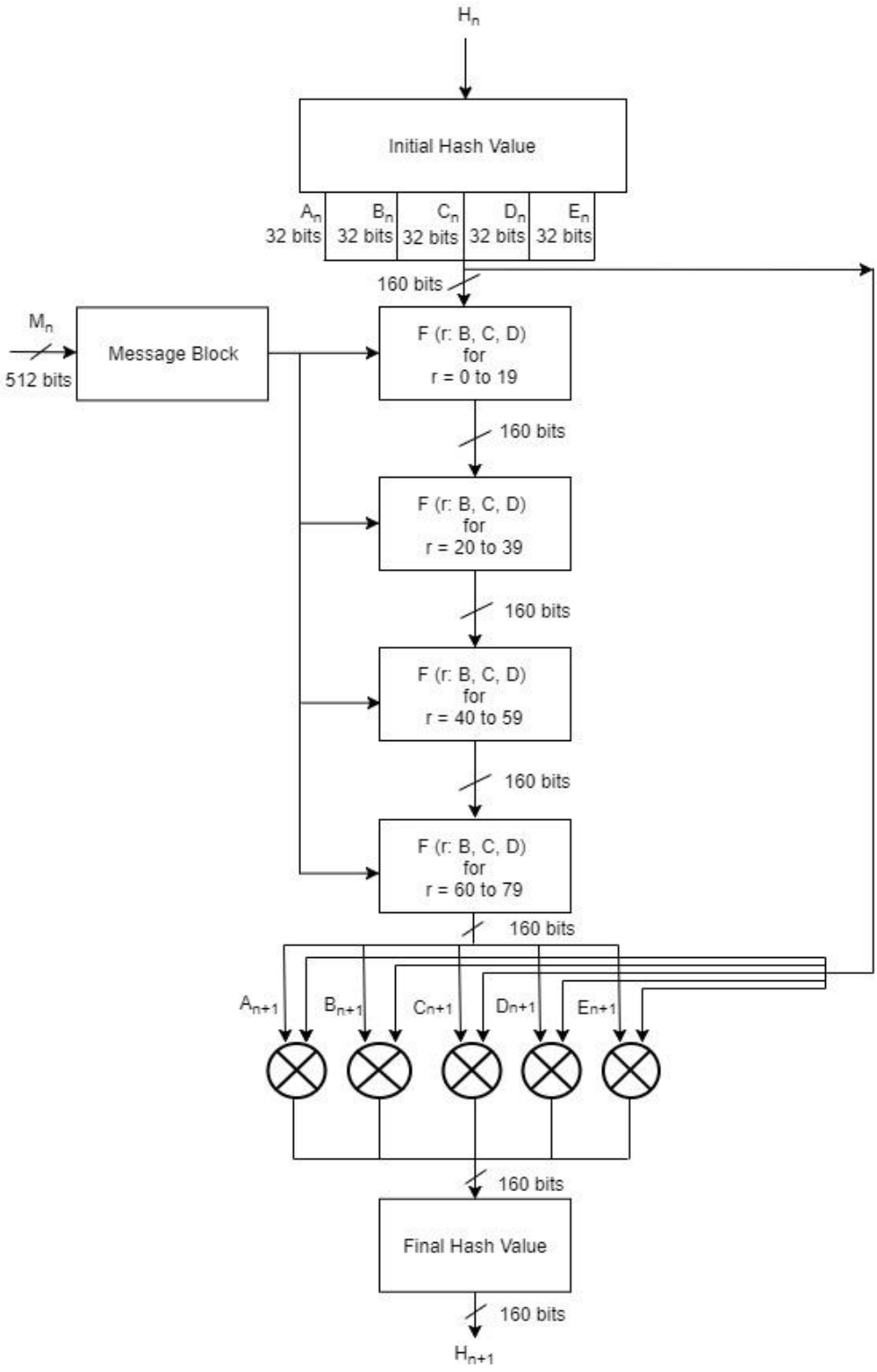


Figure 8: Block Diagram for SHA1 Processing Function

2.2 WPA2-PSK Data Confidentiality

WPA2-PSK uses Advanced Encryption Standard (AES) cipher for data confidentiality. The AES algorithm is a symmetric block cipher that can encrypt and decrypt information. Encryption converts data to an unintelligible form called ciphertext and decryption converts the ciphertext back into its original form, called plaintext [14]. AES has input block size of 128 bits and key size can be of 128, 192 or 256 bits. ***WPA2-PSK*** uses key size of 256 bits i.e. ***AES-256***. It requires 60 rounds for key expansion and the size of the expanded key is 240 bytes. When decryption is performed using Equivalent Inverse Cipher method, then there are separate set of expanded keys for encryption and decryption processes [14]. Hence, in total, there will be 480 bytes of expanded keys when decryption is performed using Equivalent Inverse Cipher method. AES-256 requires 14 rounds each for the completion of encryption and decryption processes and the size of output block is 128 bits.

AES algorithm's operations are performed on a two-dimensional array of bytes called the State [14]. For AES-256, the State consists of four rows of bytes, each containing 4 bytes. At the start of the encryption and decryption, the input – the array of bytes In0, In1, … In15 – is copied into the State array as illustrated in Figure 9. The encryption and decryption operations are then conducted on this State array, after which its final value is copied to the output – the array of bytes Out0, Out1, … Out15 as shown in Figure 10. [14].

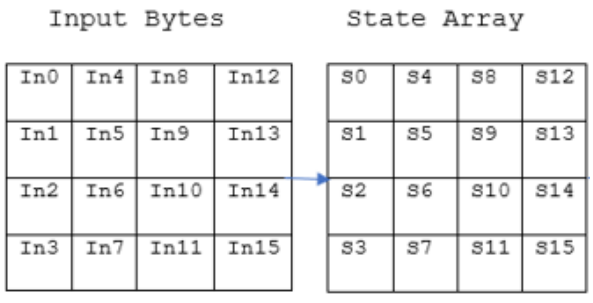


Figure 9: Input Bytes Arranged in State Array at Beginning of AES Operation

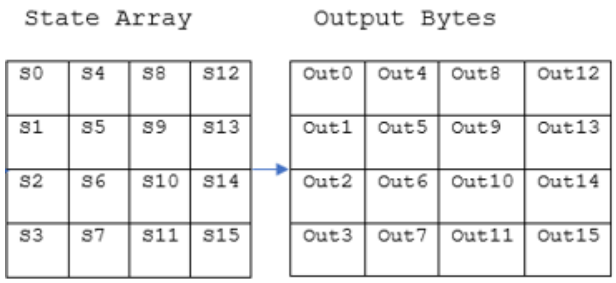


Figure 10: Output Bytes Arranged from State Array at End of AES Operation

AES consists of key expansion, encryption and decryption processes. These processes are realized with the help of following transformations [14]:

- **`RotWord ():`** This function takes 4 bytes as an argument. It performs circular left shift on the 4 input bytes.

  Example: `1,2,3,4 to 2,3,4,1`

- **`Rcon ():`** This function returns a 4-byte value based on Figure 11.

```
Rcon(0)     = 01000000
Rcon(1)     = 02000000
Rcon(2)     = 04000000
Rcon(3)     = 08000000
Rcon(4)     = 10000000
Rcon(5)     = 20000000
Rcon(6)     = 40000000
Rcon(7)     = 80000000
Rcon(8)     = 1B000000
Rcon(9)     = 36000000
Rcon(10)    = 6C000000
Rcon(11)    = D8000000
Rcon(12)    = AB000000
Rcon(13)    = 4D000000
Rcon(14)    = 9A000000
```

Figure 11: Rcon () Values

- **`AddRoundKey ():`** In the `AddRoundKey ()` transformation, a Round Key is added to the state by a simple bitwise XOR operation.

- **`SubBytes ():`** In this transformation, each byte of data is substituted with the corresponding value from the S-box lookup table (Table 7 in Appendix A).
- **`ShiftRows ():`** This function arranges the bytes of the state in 4x4 matrix and performs byte-wise circular left shift. The order of the shift varies with rows. The shift operation is not performed for the first row. The second row is shifted by 1 byte, the third row is shifted by 2 bytes and the fourth row is shifted by 3 bytes. An example of shift row operation is shown in Figure 12.

Figure 12: Example of ShiftRows ()

- **`MixColumns ():`** The matrix obtained from the `ShiftRows ()` operation goes through the multiplication over Galois Field (Figure 13). The lookup tables required for multiplication over Galois Field in the `MixColumns ()` operation are shown in Table 9 and Table 10 in Appendix A.

```
b0  = mul2[state[0]] XOR mul3[state[1]] XOR state[2] XOR state[3]
b1  = state[0] XOR mul2[state[1]] XOR mul3[state[2]] XOR state[3]
b2  = state[0] XOR state[1] XOR mul2[state[2]] XOR mul3[state[3]]
b3  = mul3[state[0]] XOR state[1] XOR state[2] XOR mul2[state[3]]
b4  = mul2[state[4]] XOR mul3[state[5]] XOR state[6] XOR state[7]
b5  = state[4] XOR mul2[state[5]] XOR mul3[state[6]] XOR state[7]
b6  = state[4] XOR state[5] XOR mul2[state[6]] XOR mul3[state[7]]
b7  = mul3[state[4]] XOR state[5] XOR state[6] XOR mul2[state[7]]
b8  = mul2[state[8]] XOR mul3[state[9]] XOR state[10] XOR state[11]
b9  = state[8] XOR mul2[state[9]] XOR mul3[state[10]] XOR state[11]
b10 = state[8] XOR state[9] XOR mul2[state[10]] XOR mul3[state[11]]
b11 = mul3[state[8]] XOR state[9] XOR state[10] XOR mul2[state[11]]
b12 = mul2[state[12]] XOR mul3[state[13]] XOR state[14] XOR state[15]
b13 = state[12] XOR mul2[state[13]] XOR mul3[state[14]] XOR state[15]
b14 = state[12] XOR state[13] XOR mul2[state[14]] XOR mul3[state[15]]
b15 = mul3[state[12]] XOR state[13] XOR state[14] XOR mul2[state[15]]
```

Figure 13: MixColumns () Calculation

- **`InvSubBytes ():`** In this transformation, each byte of data is substituted with the corresponding value from the inverse S-box table (Table 8 in Appendix A).
- **`InvShiftRows ():`** This function arranges the bytes of the state in 4x4 matrix and performs byte-wise circular right shift. The order of the shift varies with rows. The shift operation is not performed for the first row. The second row is shifted by 1 byte, the third row is shifted by 2 bytes and the fourth row is shifted by 3 bytes. An example of shift row is shown in Figure 14 [15].

Figure 14: Example if InvShiftRows ()

- **`InvMixColumns ():`** The matrix obtained from the `InvShiftRows ()` operation goes through the multiplication over Galois Field (Figure 15). The lookup tables required for multiplication over Galois Field in the `InvMixColumns ()` operation are shown in Table 11, Table 12, Table 13 and Table 14 in Appendix A

```
b0  = mul14[state[0]] XOR mul11[state[1]] XOR mul13[state[2]] XOR mul9[state[3]]
b1  = mul9[state[0]] XOR mul14[state[1]] XOR mul11[state[2]] XOR mul13[state[3]]
b2  = mul13[state[0]] XOR mul9[state[1]] XOR mul14[state[2]] XOR mul11[state[3]]
b3  = mul11[state[0]] XOR mul13[state[1]] XOR mul9[state[2]] XOR mul14[state[3]]
b4  = mul14[state[4]] XOR mul11[state[5]] XOR mul13[state[6]] XOR mul9[state[7]]
b5  = mul9[state[4]] XOR mul14[state[5]] XOR mul11[state[6]] XOR mul13[state[7]]
b6  = mul13[state[4]] XOR mul9[state[5]] XOR mul14[state[6]] XOR mul11[state[7]]
b7  = mul11[state[4]] XOR mul13[state[5]] XOR mul9[state[6]] XOR mul14[state[7]]
b8  = mul14[[state[8]] XOR mul11[state[9]] XOR mul13[state[10]] XOR mul9[state[11]]
b9  = mul9[state[8]] XOR mul14[state[9]] XOR mul11[state[10]] XOR mul13[state[11]]
b10 = mul13[state[8]] XOR mul9[state[9]] XOR mul14[state[10]] XOR mul11[state[11]]
b11 = mul11[state[8]] XOR mul13[state[9]] XOR mul9[state[10]] XOR mul14[state[11]]
b12 = mul14[state[12]] XOR mul11[state[13]] XOR mul13[state[14]] XOR mul9[state[15]]
b13 = mul9[state[12]] XOR mul14[state[13]] XOR mul11[state[14]] XOR mul13[state[15]]
b14 = mul13[state[12]] XOR mul9[state[13]] XOR mul14[state[14]] XOR mul11[state[15]]
b15 = mul11[state[12]] XOR mul13[state[13]] XOR mul9[state[14]] XOR mul14[state[15]]
```

Figure 15: InvMixColumns Calculation

2.2.1 AES-256 Key Expansion

The AES algorithm takes the Cipher Key and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of `Nb (Nr + 1)` words: the algorithm requires an initial set of `Nb` words, and each of the `Nr` rounds requires `Nb` words of key data [14]. The resulting key schedule consists of a linear array of 4-byte words, denoted [`wi`], with `i` in the range `0< i< Nb (Nr + 1)`) [14].

In ***AES-256***, `Nb` = 4, `Nk` = 8 and `Nr` = 14, so the key expansion routine generates a total of 60 words (240 bytes). The key expansion routine runs for 14 rounds and generates 240 bytes

of expanded key. Two different sets of 240 bytes of expanded keys are generated for encryption and decryption when the decryption is done using Equivalent Inverse Cipher method. The expansion of the input key into the key expansion routine proceeds according to the pseudo code in Figure 16 [14]. For ***AES-256***, `Nb` = 4, `Nk` = 8 and `Nr` = 14.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
    word temp
    i = 0
    while (i < Nk)
        w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
        i = i+1
    end while
    i = Nk
    while (i < Nb * (Nr+1)]
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if (Nk > 6 and i mod Nk = 4)
            temp = SubWord(temp)
        end if
        w[i] = w[i-Nk] xor temp
        i = i + 1
    end while
end
```

Figure 16: Pseudo Code for Key Expansion for Encryption

For the Equivalent Inverse Cipher, the following pseudo code shown in Figure 17 must be added to the end of the pseudo code shown in Figure 16 [14]. For ***AES-256***, `Nb` = 4, `Nk` = 8 and `Nr` = 14.

```
for i = 0 step 1 to (Nr+1)*Nb-1
    dw[i] = w[i]
end for
for round = 1 step 1 to Nr-1
    InvMixColumns(dw[round*Nb, (round+1)*Nb-1]) // note change of type
end for
```

Figure 17: Additional Pseudo Code to be Added for Key Expansion for Decryption

2.2.2 AES-256 Encryption

In each round of AES encryption, the cipher makes four different transformations to the block of data. The four transformations are: `AddRoundKey` (), `SubBytes` (), `ShiftRows` () and `MixColumns` (). The exception is the final round, which only has three transformations since it does not have the `MixColumns ()` operation [16]. During these rounds, each block of data is depicted as 4 x 4 byte matrix and the key is divided into 4 x 4 byte matrix as well. Each round gets these matrices as an input and produces 4 x 4 byte state matrix as an output.

The pseudo code for AES encryption is shown in Figure 18. For ***AES-256***, `Nb` = 4, `Nk` = 8 and `Nr` = 14.

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, w[0, Nb-1])
    for round = 1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
    out = state
end
```

Figure 18: Pseudo Code for AES Encryption

2.2.3 AES-256 Decryption

Like encryption, in each round of decryption, the cipher makes four different transformations to the block of data. The four transformations being: `InvAddRoundKey` (), `InvSubBytes` (), `InvShiftRows` () and `InvMixColumns` (). The exception is the final

round, which only has three transformations since it does not have the `InvMixColumns ()` operation [16]. Similarly, during these rounds, each block of data is arranged as 4 x 4 byte matrix and the key is also arranged as 4 x 4 byte matrix. But the expanded keys used here are different to the ones used for encryption. Each round gets these matrices as an input and produces 4 x 4 byte state matrix as an output. Pseudo code for the Equivalent Inverse Cipher is shown in Figure 19 [14]. For ***AES-256***, `Nb` = 4, `Nk` = 8 and `Nr` = 14.

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
begin
    byte state[4,Nb]
    state = in
    AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])
    for round = Nr-1 step -1 downto 1
        InvSubBytes(state)
        InvShiftRows(state)
        InvMixColumns(state)
        AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
    end for
    InvSubBytes(state)
    InvShiftRows(state)
    AddRoundKey(state, dw[0, Nb-1])
    out = state
end
```

Figure 19: Pseudo Code for the Equivalent Inverse Cipher

The overall block diagram of ***AES-256*** is shown in Figure 20.

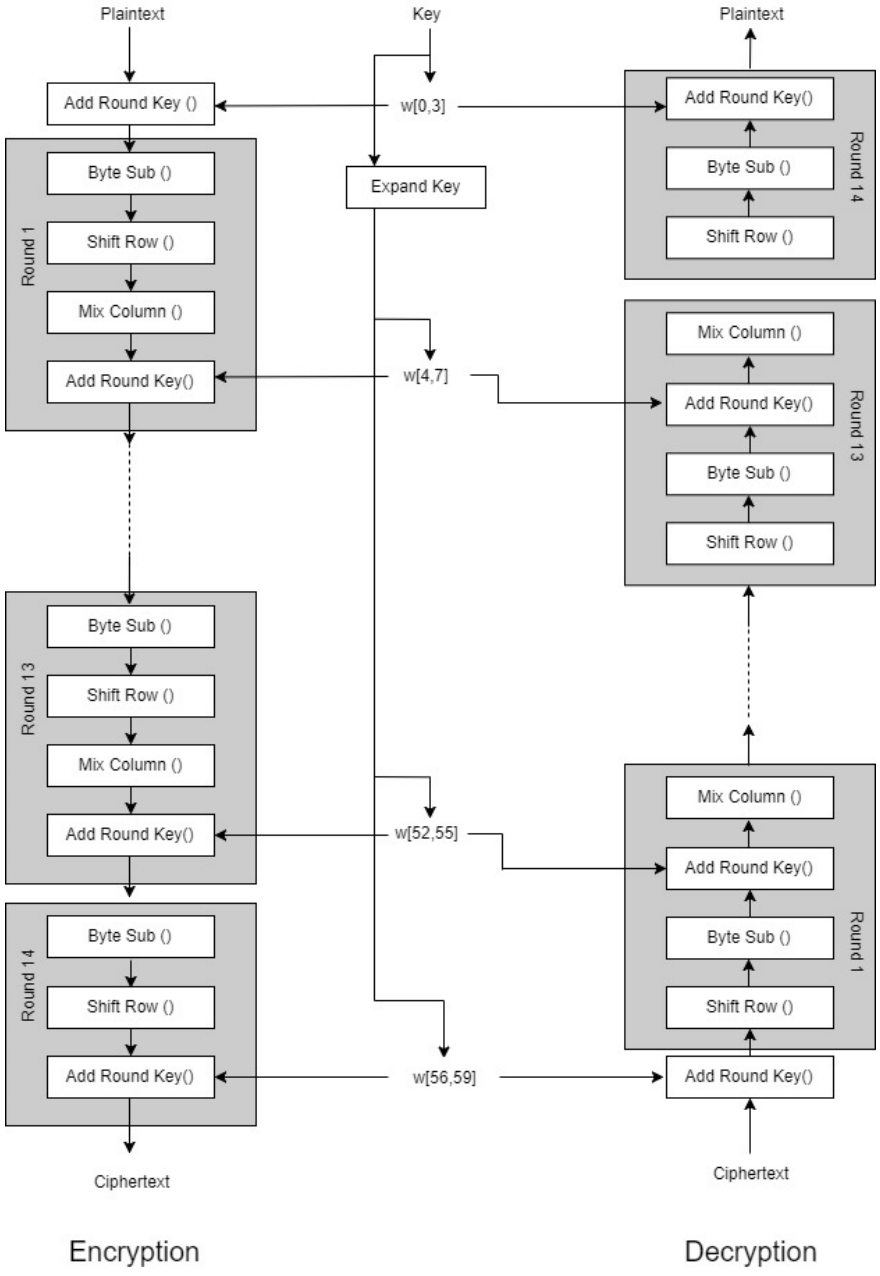


Figure 20: AES-256 Block Diagram

Chapter 3 explains how the theory related to ***PBKDF2***, ***HMAC*** and ***SHA1*** used in ***WPA2-PSK*** device authentication, and key expansion used in ***AES-256*** encryption and decryption were implemented in software using C programming language.

Chapter 3: Software Implementation

3.1 Overview

Derivation of PMK using ***PBKDF2*** for authentication, and key expansion for encryption and decryption for ***AES-256***, were implemented in software. This was because authentication is only performed once per node before the ***Wi-Fi*** connection has been established. Hence, ***PBKDF2***, ***HMAC*** and ***SHA1*** were performed only for the authentication process. After authentication, these operations didn't have to be repeatedly used during the data communication phase. Similarly, key expansion for ***AES-256*** is only performed once. After expansion, keys were stored in the memory and retrieved only when they were required.

All software development in this thesis was implemented using combination of C and C++ programming languages for the MicroBlaze™ embedded processor soft core in Xilinx Software Development Kit (SDK). Code related to ***PBKDF2*** and ***AES-256*** key expansion was written in C, while the application code was written in C++.

3.2 MicroBlaze™ Environment

The MicroBlaze™ embedded processor soft core is a reduced instruction set computer (RISC) optimized for implementation in Xilinx® Field Programmable Gate Arrays (FPGAs). Figure 21 shows a functional block diagram of the MicroBlaze core [17].

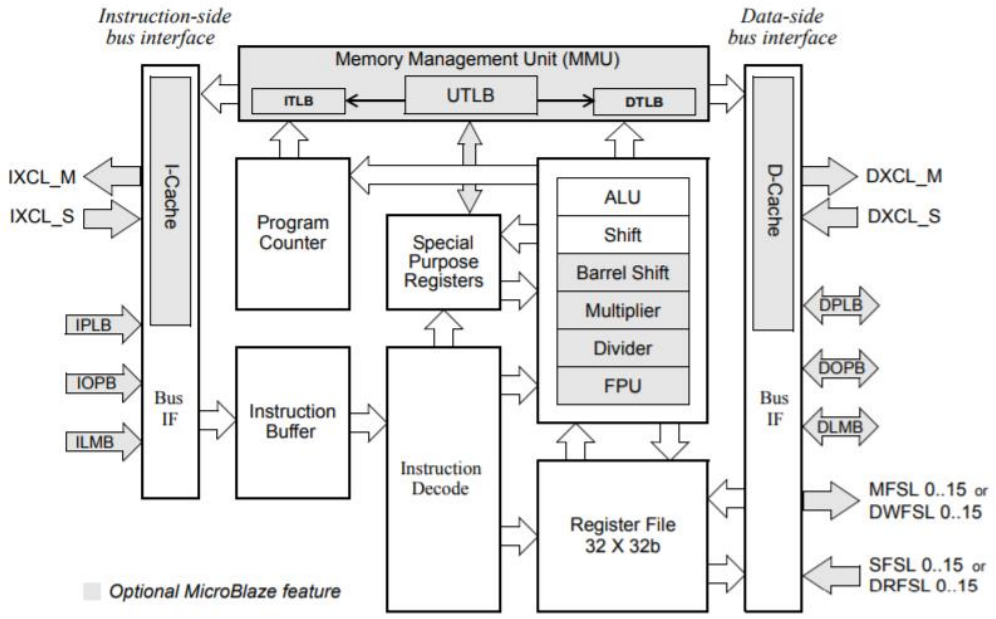


Figure 21: MicroBlaze Core Block Diagram

The MicroBlaze soft core processor is highly configurable, allowing you to select a specific set of features required by your design. The fixed feature set of the processor includes [17].

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline

MicroBlaze core is organized as a Harvard architecture with separate bus interface units for data accesses and instruction accesses [18]. It does not separate between data accesses to I/O and memory (i.e. it uses memory mapped I/O). The processor has up to three interfaces for memory accesses [18]: Local Memory Bus (LMB), IBM's On-chip Peripheral Bus (OPB), and Xilinx CacheLink (XCL). MicroBlaze also supports reset, interrupt, user exception, break and hardware exceptions. For interrupts, MicroBlaze supports only one external interrupt source

(connecting to the Interrupt input port) [18]. If multiple interrupts are needed, an interrupt controller must be used to handle multiple interrupt requests to MicroBlaze. The stack convention used in MicroBlaze starts from a higher memory location and grows downward to lower memory locations when items are pushed onto a stack with a function call [18]. Items are popped off the stack the reverse order they were put on. Writing software to control the MicroBlaze processor must be done in C/C++ language [18].

3.3 WPA2-PSK Device Authentication

The code for ***WPA2-PSK*** authentication was written using C programming language. The overall code implementation contained 4 layers. At the bottom layer, there was code for ***SHA1*** Hash algorithm. The second layer of code was for ***HMAC-SHA1***, which called functions from the first layer. The third layer of code was for ***PBKDF2*** which would call functions from the second layer. Finally, the application layer called the functions in the ***PBKDF2*** layer for the complete ***WPA2-PSK*** functionality. This layered architecture is illustrated in Figure 22.

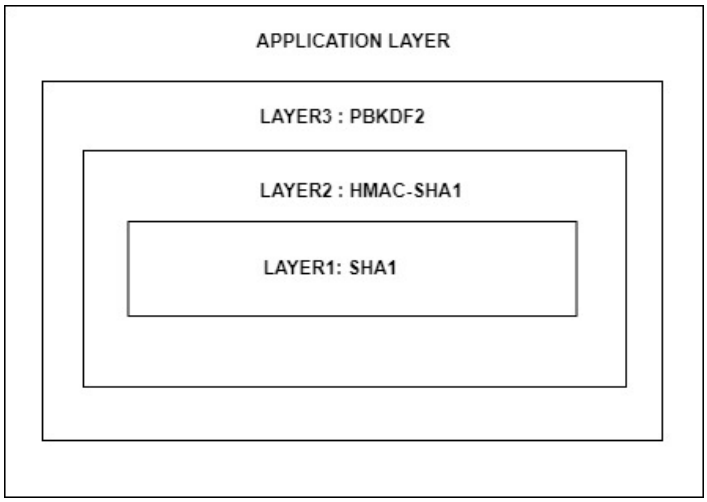


Figure 22: Layered Software Implementation of WPA2-PSK Authentication

3.3.1 Layer1: SHA1-HASH Implementation

This layer dealt with all the functions related to the implementation of ***SHA1*** Hash algorithm. The information regarding relevant data types and the function prototypes for all the functions in this layer are given below:

- **Data Type**: `SHA1_CTX`

```
typedef struct {
    uint8_t data [64];
    uint32_t datalen;
    unsigned long long bitlen;
    uint32_t state [5];
} SHA1_CTX;
```

This data type was used to maintain information relevant to an iteration of an ***SHA1***-Hash process. It held information regarding 512-bits of input block, 160-bits of output hash, data length and bit length. This data type was used to pass information to and store information from all the functions defined in the ***SHA1***-Hash Layer.

- **Function Prototype 1**: `SHA1Init ()`

```
void SHA1Init (SHA1_CTX *context)
```

This Function was used to initialize a new context for the ***SHA1***-Hash process. It initialized `datalen` and `bitlen` to `0`. It also initialized the states of the context to the initial Hash values of `0x67452301 ,0xEFCDAB89, 0x98BADCFE, 0x10325476 and 0xC3D2E1F0.`

- **Function Prototype 2**: `SHA1Update ()`

```
void SHA1Update (SHA1_CTX *context, const void *data, uint32_t len)
```

This function was used to update `data`, `data length` and `bit length` for the context of ***SHA1***. If the `data length` was 64 bytes (512 bits) i.e. input block size of ***SHA1***, we started the ***SHA1***-Hash process by calling `SHA1Transform ()`.

- **Function Prototype 3**: `SHA1Final ()`

```
void SHA1Final (unsigned char digest [20], SHA1_CTX* context)
```

This was the function where the initial processing before the actual ***SHA1***-Hash processing was done. The initial padding and the appending of the data length to the input block was done in this function. After the initial processing, it called `SHA1Transform ()` to perform the ***SHA1***-Hash algorithm.

- **Function Prototype 4**: `SHA1Transform ()`

```
void SHA1Transform (SHA1_CTX *ctx, const uint8_t data [])
```

This was the main function where the processing part of the actual ***SHA1***-Hash Algorithm was implemented. First, the 32-bit words of the 512-bit input block were stored into initial 16 arrays of size 32-bits. `W (16)` to `W (79)` values were then calculated from these values. 80 rounds of ***SHA1***- Hash transform was performed on the data to get the 160-bits of ***SHA1***-Hash Output value. Finally, the new Hash for the context was updated with the new output Hash value.

3.3.2 Layer2: HMAC_SHA1 Implementation

This layer dealt with the function related to the implementation of ***HMAC-SHA1***. It contained a single function that called functions defined in ***SHA1***-Hash algorithm. The prototype for this function is given below:

```
void my_hmac_sha1(const unsigned char *text,
                  int text_len, const unsigned char *key,
                  int key_len, unsigned char *digest)
```

This single function was used for the ***HMAC*** operation over ***SHA1***-Hash. The first step of the code performed the initial padding and appending operations required for ***HMAC*** operation. After this initial process, this function would successively call `SHA1Init ()`. `SHA1Update (), SHA1Final () and SHA1Transform ().`

3.3.3 Layer3: PBKDF2 Implementation

This layer dealt with the function related to the implementation of ***PBKDF2*** operation. It contained a single function that called `my_hmac_sha1()` for 4096 iterations after some initial processing. The prototype for this function is given below:

```
int pkcs5_pbkdf2(const char *pass, const uint8_t *salt, size_t salt_len,
                 unsigned int rounds, uint8_t *key, size_t key_len)
```

256-bit PMK was derived from this function which was used as the first key in AES encryption. The application code would use this function to obtain the PMK from SSID-Passphrase combination, which is used to authenticate a wireless device using ***WPA2-PSK*** before starting the wireless communication.

3.4 AES-256 Key Expansion

The code for ***AES-256*** key expansion was written using C programming language and for 32-bit soft processor MicroBlaze. The information regarding relevant function definitions for all the functions used for key expansion are given below:

- **Function Definition 1**: `RotWord ()`

```
static void RotWord(uint8_t * value){
    uint8_t temp;
    temp = value[0];
    value[0] = value[1];
    value[1] = value[2];
    value[2] = value[3];
    value[3] = temp;
}
```

This function took an array which contained 4 one-byte values [a0, a1, a2, a3] as input, performed a cyclic permutation, and returned [a1, a2, a3, a0] as the output.

- **Function Definition 2**: `SubWord ()`

```
static void SubWord(uint8_t * value){
    value[0] = sbox[value[0]];
    value[1] = sbox[value[1]];
    value[2] = sbox[value[2]];
    value[3] = sbox[value[3]];
}
```

This function took an array which contained 4 one-byte values [a0, a1, a2, a3] as input and returned S-Box substituted values of the array as the output. The code snippet for S-Box lookup table implementation in C is shown in Figure 48 (Appendix B).

- **Function Definition 3**: `InvMixColumns ()`

```
static uint8_t InvMixColumns(uint8_t * invkey, uint8_t count)
{
    int i;
    unsigned char a,b,c,d;
    for(i=0;i<4;i++)
    {

        a = state[0][i];
        b = state[1][i];
        c = state[2][i];
        d = state[3][i];

        invkey[count++] = state[0][i] = mul14[a] ^ mul11[b] ^ mul13[c] ^ mul9[d];
        invkey[count++] = state[1][i] = mul9[a] ^ mul14[b] ^ mul11[c] ^ mul13[d];
        invkey[count++] = state[2][i] = mul13[a] ^ mul9[b] ^ mul14[c] ^ mul11[d];
        invkey[count++] = state[3][i] = mul11[a] ^ mul13[b] ^ mul9[c] ^ mul14[d];
    }
    return count;
}
```

This function took an array containing 16 one-byte values [a0, a1, a2 … a15] and number of bytes of completed expanded key for equivalent inverse cipher as an input, performed Inverse Mix Column transformations, and returned the transformed values and new count of the completed expanded key as the output. In this function definition, `state` was a global 4x4 one-byte array. The code snippets for mul9, mul11, mul13 and mul14 lookup tables implementations in C are shown in Figure 49, Figure 50, Figure 51 and Figure 52 respectively in Appendix B.

Function Definition 4: `KeyExpansion ()`

```
static void KeyExpansion( uint8_t* KeyIn, uint8_t* RoundKey, uint8_t *InvRoundKey){
  unsigned i, j, k;
  uint8_t tempa[4];
  for (i = 0; i < Nk; i++){
    RoundKey[(i * 4) + 0] = KeyIn[(i * 4) + 0];
    RoundKey[(i * 4) + 1] = KeyIn[(i * 4) + 1];
    RoundKey[(i * 4) + 2] = KeyIn[(i * 4) + 2];
    RoundKey[(i * 4) + 3] = KeyIn[(i * 4) + 3];
  }
  for (i = Nk; i < Nb * (Nr + 1); ++i){
        k = (i - 1) * 4;
        tempa[0]=RoundKey[k + 0];
        tempa[1]=RoundKey[k + 1];
        tempa[2]=RoundKey[k + 2];
        tempa[3]=RoundKey[k + 3];
    if (i % Nk == 0){
        RotWord(tempa);
        SubWord(tempa);
        tempa[0] = tempa[0] ^ Rcon[i/Nk];
    }else if (i % Nk == 4){
        SubWord(tempa);
    }
    j = i * 4; k=(i - Nk) * 4;
    RoundKey[j + 0] = RoundKey[k + 0] ^ tempa[0];
    RoundKey[j + 1] = RoundKey[k + 1] ^ tempa[1];
    RoundKey[j + 2] = RoundKey[k + 2] ^ tempa[2];
    RoundKey[j + 3] = RoundKey[k + 3] ^ tempa[3];
  }
  /* Code for key expansion for equivalent inverse cipher starts */
  for(k=0;k<16;k++){
    InvRoundKey[woutCount++] = RoundKey[k];
  }
  while(k <224){
      for(i=0;i<4;i++){
        for(j=0;j<4;j++){
            state[j][i] =  RoundKey[k++];
        }
      }
      woutCount = InvMixColumns(wout,woutCount);
  }
  for(;k<240;k++){
    InvRoundKey[woutCount++] = RoundKey[k];
  }
}
```

This function took an array which contained 32 bytes of PMK as input, expanded it to 240 bytes each of expanded key for encryption and decryption, and returned these expanded keys as the output. These expanded keys were stored in BRAM of the FPGA.

Chapter 4 describes how these keys were arranged in the BRAM after the expansion process, retrieved from the BRAM, and how they would be used in the hardware implementation of the ***AES-256*** module to encrypt and decrypt data. Chapter 4 also shows how the encrypted/decrypted data were stored in BRAM after the transformation.

Chapter 4: Hardware Implementation

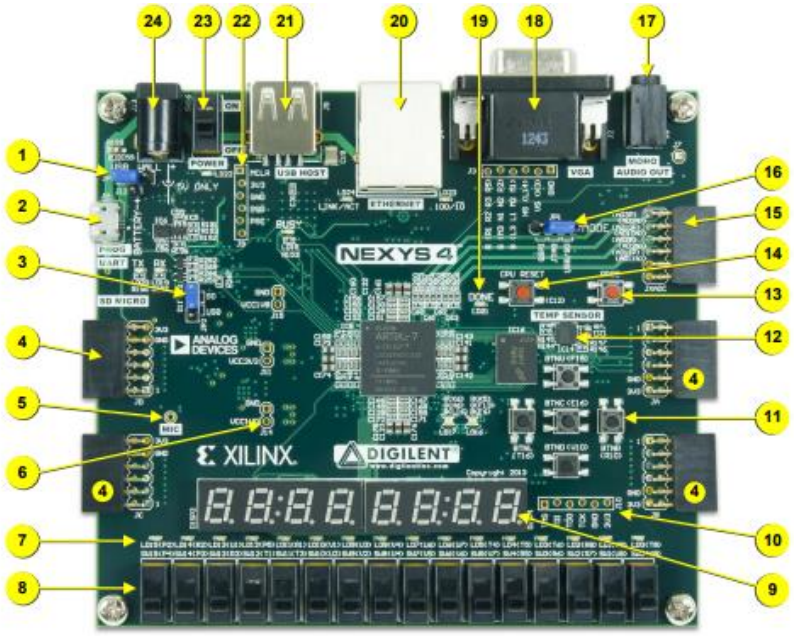
4.1 Overview

Encryption and decryption on data with ***AES-256*** module was implemented on hardware using Verilog HDL. This was because, after authentication, all data communicated within the network had to be encrypted. And, we used the AES-256 core to encrypt and decrypt the data being communicated within the network. Since, this core must be used repeatedly for data confidentiality during communication, it was implemented in hardware. The Verilog Code was written on Xilinx Vivado Design Suite 2017.2 IDE. After ***AES-256*** module designed using Verilog HDL was synthesized and implemented, the design was packaged into an IP core using Vivado IP Packager tool. This ***AES-256*** IP core was then interfaced to MicroBlaze softcore processor with the help of AXI Interconnect. This design was targeted for Nexys4 FPGA board. The reason for using Nexys4 board was that MicroBlaze softcore processor needs a lot of FPGA resources. Additionally, further FPGA resources were also required in the hardware implementation of AES-256. Since, Nexys4 board was able to support these requirements, it was used as the target device for this thesis.

4.2 Nexys4

Nexys4 board is a development platform based on the latest Artix-7™ (Xilinx part number XC7A100T-1CSG324C) Field Programmable Gate Array (FPGA) from Xilinx [19]. The Artix-7 FPGA is designed for high performance and it features 15850 logic slices (each with 6-input LUTs and 8 flip-flops), 240 DSP slices and 4860 KB of fast block RAM [20]. Nexys4 has generous external memories, and collection of USB, Ethernet, and other ports, and can host designs ranging from introductory combinational circuits to powerful embedded processors [19].

It also has several built-in peripherals such as an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and a lot of I/O devices [19]. Nexys4 board with its component description is shown in Figure 23.



Callout	Component Description	Callout	Component Description
1	Power select jumper and battery header	13	FPGA configuration reset button
2	Shared UART/ JTAG USB port	14	CPU reset button (for soft cores)
3	External configuration jumper (SD / USB)	15	Analog signal Pmod port (XADC)
4	Pmod port(s)	16	Programming mode jumper
5	Microphone	17	Audio connector
6	Power supply test point(s)	18	VGA connector
7	LEDs (16)	19	FPGA programming done LED
8	Slide switches	20	Ethernet connector
9	Eight digit 7-seg display	21	USB host connector
10	JTAG port for (optional) external cable	22	PIC24 programming port (factory use)
11	Five pushbuttons	23	Power switch
12	Temperature sensor	24	Power jack

Figure 23: Nexys4 Board Features

4.3 AXI Interconnect

Advanced extensible Interface (AXI) is a part of the Arm Advanced Microcontroller Bus Architecture (AMBA) specification that provides the interface between the processing system

and programmable logic sections of the chip [21]. The AXI specifications describe an interface between a single AXI master and a single AXI slave, representing IP cores that exchange information with each other [22]. Memory mapped AXI masters and slaves can be connected using a structure called an Interconnect block [22]. The Xilinx AXI Interconnect IP contains AXI-compliant master and slave interfaces and can be used to route transactions between one or more AXI masters and slaves [22] . AXI Interconnect connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices [23]. When connecting one master to one slave, the AXI Interconnect core can perform address range checking. Also, it can perform any of the normal data-width, clock rate, or protocol conversions and pipelining [23]. When not performing any conversions or address range checking, the AXI Interconnect core is implemented as wires, with no resources, no delay and no latency [23].

4.4 AES-256 Implementation

This hardware implementation of ***AES-256*** primarily contained 2 components: BRAMs (Key BRAM and Data BRAM) and AES core. BRAMs were used to store expanded keys, plaintext and ciphertext, whereas, the AES core had modules for encryption and decryption blocks that used the expanded key, plaintext and ciphertext for encryption and decryption.

Since block size of AES is 128 bits (16 Bytes), both encryption and decryption processes worked on 128-bits (16 Bytes) of data at a time. If the data to be transformed was less than 128 bits, then, it was padded with trailing 0’s to make it 128-bit block before being transformed. If the data to be transformed was greater than 128 bits but not a multiple of 128 bits, then, it was also padded with trailing 0’s until we had a data block which is multiple of 128 bits. After padding, the encryption/decryption was done on one 128-bit block of data at a time. The data to

be transformed was initially stored in a certain location of Data BRAM. At the beginning of the transformation, they were read from the Data BRAM and passed to the AES core. After the transformation was completed, the converted data was stored in a separate location of the Data BRAM that was allocated for the transformed data.

4.4.1 BRAMs

BRAMs are blocks of 32-bit memory locations used to store expanded key, original data to be encrypted or decrypted, and transformed data after encryption or decryption. It is a synchronous memory block with 32-bit data being clocked in or out at every clock. Two instances of BRAMs were created for the implementation of ***AES-256*** module: Key BRAM and Data BRAM.

- **Key BRAM**: The Data width of the Key BRAM was 32-bits. The expanded keys were stored in Key BRAM. 4 Bytes of expanded keys were stored per BRAM location. Hence, 60 memory locations were used to store 240 bytes of expanded key for encryption. Similarly, further 60 memory locations were used to store the other 240 bytes of expanded key for decryption. These keys were only read once from the BRAM at the beginning of the encryption/decryption process and saved into a temporary buffer. For multiple blocks of encryption/decryption, the keys were accessed directly from the temporary buffer instead of BRAM. The start address for Key BRAM was 0xC2000000. The overall memory organization for Key BRAM locations used in the design of ***AES-256*** module is shown in Figure 24.

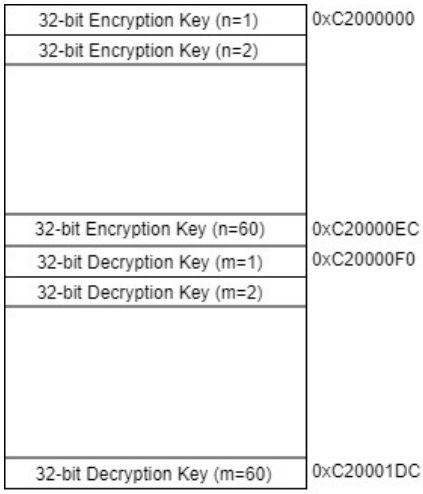


Figure 24: Key BRAM Organization

- **Data BRAM**: It was a Read/Write Memory block where original data and data to be transformed were stored. The Data width of the Data BRAM was 32-bits. 32 such memory locations were allocated each for original data and transformed data. Hence, 128 bytes of BRAM memory was used for storing original and transformed data. The start address for Data BRAM was 0xC0000000. The overall memory organization for Data BRAM is shown in Figure 25.

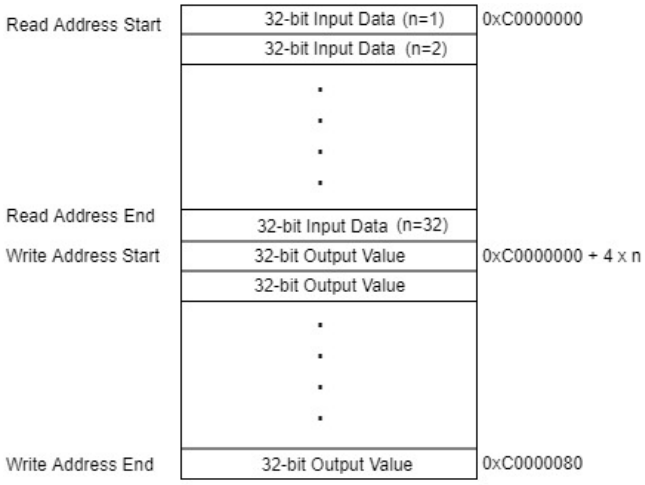


Figure 25: Data BRAM Organization

4.4.2 AES Core

The block diagram of top level of ***AES-256*** core implementation is shown in Figure 26. It has the following port definitions:

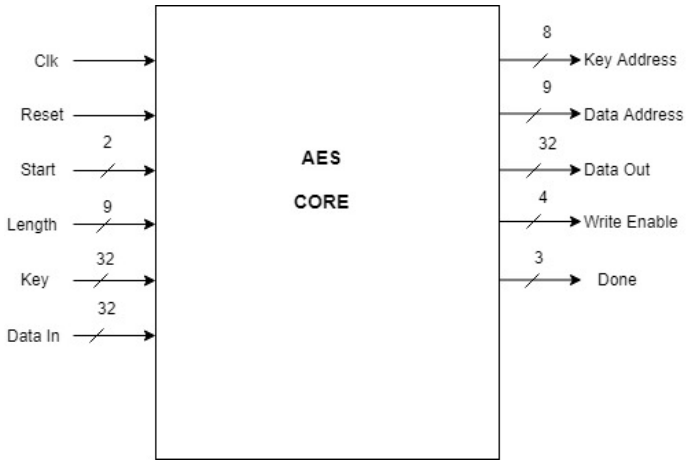


Figure 26: AES Core with Input and Output Signals

- **Input Ports**:
    - **Clk**: It was the clock source to the AES core. The frequency of clock used in the design was 100 Mhz.
    - **Reset**: It was the synchronous reset signal to the AES core. It worked on active low logic. The reset signal was controlled by bit 11 of Slave register0 from the AXI BUS of MicroBlaze.
    - **Start**: It was a 2-bit wide input signal which was used to start the encryption or decryption signal. If **Start** = 2'b01, the AES core would perform the encryption operation, if **Start** = 2'b10, the AES core would perform the decryption operation. All other possible values of **Start** signal were DON'T CARE cases.

Start signal was controlled by bits [1:0] of Slave register0 from the AXI BUS of MicroBlaze.

- **Length**: It was a 9-bit input value which signified the number of 32-bit input data (after zero padding to make it multiple of 128-bit) to be either encrypted or decrypted. The **Length** signal was controlled by bits [10:2] of Slave register0 from the AXI BUS of MicroBlaze.
- **Key**: It was a 32-bit expanded key input to the AES core. This input port was connected to 32-bit Data output port of Key BRAM.
- **Data In**: It was a 32-bit plaintext or ciphertext to be encrypted or decrypted respectively. This input port was connected to 32-bit Data output port of Data BRAM.

- **Output Ports**
  - **Key Address**: It was a 32-bit output value which denoted the memory location of the Key BRAM from which the expanded key was to be retrieved during the encryption/decryption process. On reset the value of **Key Address** was 0. This output port was connected to 32-bit address input port of Key BRAM.
  - **Data Address**: It was a 32-bit output value which denoted the memory location of the Data BRAM from which the expanded key was to be retrieved during the encryption/decryption process. On reset the value of **Data Address** was 0. This output port was connected to 32-bit address input port of Data BRAM.
  - **Data Out**: It was a 32-bit output value which signified ciphertext in case of encryption operation and plaintext in case of decryption operation. On reset the

value of **Data Out** was 0. This output port was connected to 32-bit Data input port of Data BRAM.

- **Write Enable**: It was a 4-bit output signal that was used to enable the write operation of the ciphertext in case of encryption or plaintext in case of decryption, to the allocated memory locations in the Data BRAM. On reset the value of **Write Enable** was 0. This output port was connected to 4-bit Write Enable input port of Data BRAM.
- **Done**: It was a 3-bit output signal that showed the completion of the AES operation. The value of **Done** signal would become 3'b111 after the AES transformation was completed and the new data was written into the allocated memory locations in the Data BRAM. On reset the value of **Done** was 0. This output signal was connected to bits [2:0] of Slave register1 from the AXI BUS of MicroBlaze.

This implementation took 41 clock cycles from the start signal to complete the encryption/decryption of the first 128-bit (16-Byte) block of data. After the first block of data was transformed, it only took 4 further clock cycles per block, for the other blocks of input to be encrypted or decrypted. This implementation supported transformation of 8 blocks of input data at time i.e. the depth of the input data buffer and output data buffer was 128 bytes.

4.4.3 AES-256 Internal Design

Internally, ***AES-256*** module contained several sub-modules. The internal design of ***AES-256*** with the help of these sub-modules is shown in Figure 27.

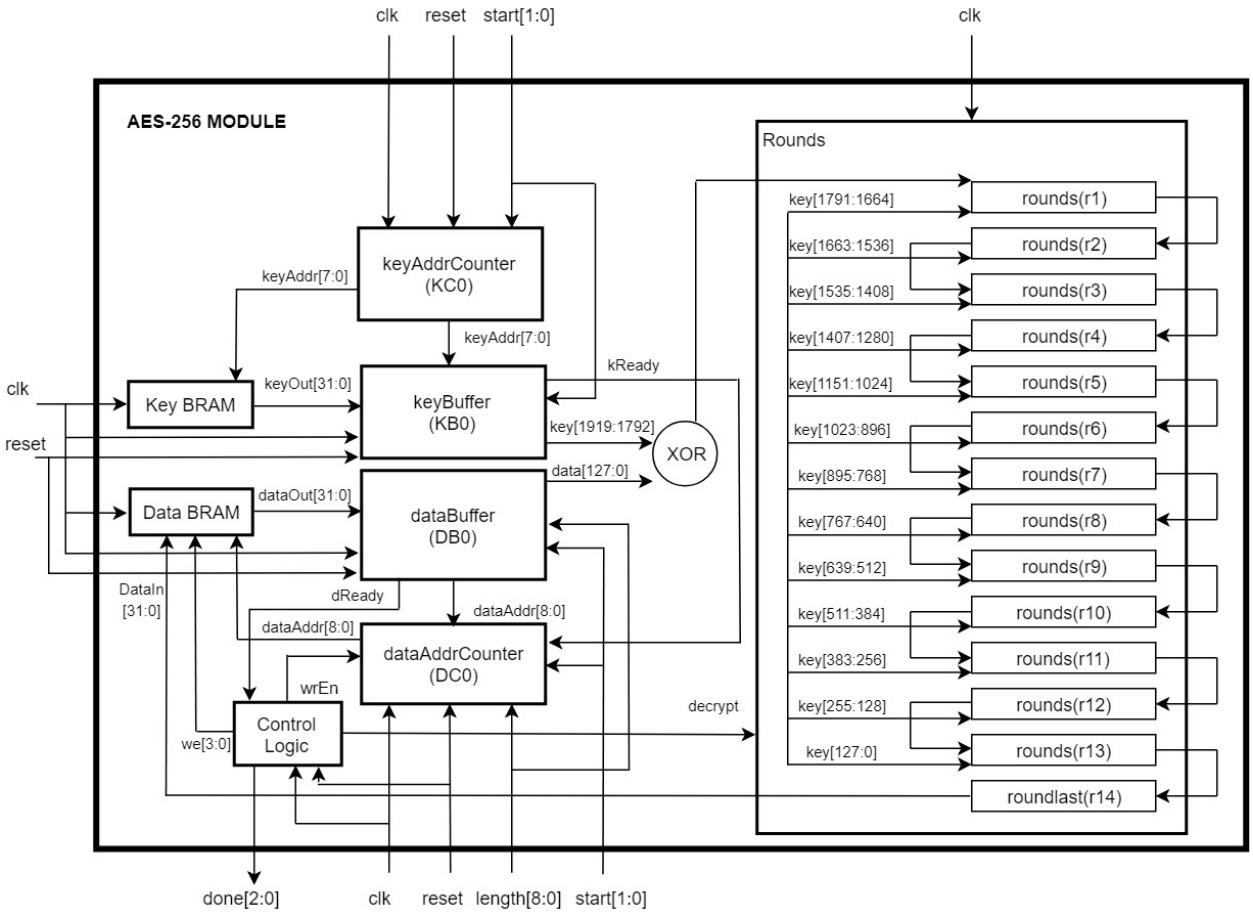


Figure 27: Internal Design of AES-256

The information regarding prototypes for various sub-modules of ***AES-256*** are given below:

- **Module Prototype 1**: `keyAddrCounter ()`

```
module keyAddrCounter(input clk, input reset ,input [1:0]start,
                      output reg [7:0]keyAddr)
```

This module was used to keep track of read address for Key BRAM. If the value of the input signal `start` was 1, then the read address would start from the memory location containing the first encryption keys. If the value of the input signal `start` was 2,

then the read address would start from the memory location containing the first decryption keys.

- **Module Prototype 2**: `keyBuffer ()`

```
module keyBuffer(input clk, input reset, input [1:0]start,
                 input [7:0]keyAddr, input [31:0]keyOut,
                 output reg [1919:0]key, output reg kReady)
```

This module was used to buffer the expanded key values from the Key BRAM to internal buffer. If the value of the input signal `start` was 1, then the module would buffer expanded keys for encryption. If the value of the input signal `start` was 2, then the module would buffer expanded keys for decryption. After the buffering is completed, it would always send a `kReady = 1` signal.

- **Module Prototype 3**: `dataAddrCounter ()`

```
module dataAddrCounter(input clk, input reset ,input start,
                       input [8:0]length,input wrEn,
                       output reg [8:0]dataAddr)
```

This module was used to keep track of read and write address for Data BRAM. If the value of the input signal `wrEn` was 0, then the read address would start from the memory location containing the first input data. If the value of the input signal `wrEn` was 1, then the write address would start from the memory location for the first output data.

- **Module Prototype 4**: `dataBuffer ()`

```
module dataBuffer(input clk, input reset, input [1:0]start,
                  input [8:0]length, input [8:0]dataAddr,
                  input [31:0]dataOut, output reg [127:0]data,
                  output reg dReady)
```

This module was used to buffer the input data from the Data BRAM to internal buffer. It could buffer up to 128 bytes of data at time. After the buffering was completed, it would send a `dReady = 1` signal.

- **Module Prototype 5**: `rounds ()`

```
module rounds(input clk, input decrypt, input [127:0]data,
              input [127:0]key, output reg [127:0]DataIn)
```

Within this module, `AddRoundKey (), SubBytes (), ShiftRows (), MixColumns (), InvAddRoundKey (), InvSubBytes (), InvShiftRows ()` and `InvMixColumns ()` transformations were implemented. In this implementation, `SubBytes ()` and `ShiftRows ()`, and, `InvSubBytes ()` and `InvShiftRows ()` were implemented inside a single block and within in a single clock. If the input signal `decrypt = 0`, then this module would perform encryption, and if `decrypt = 1`, this module would perform decryption. 13 instances of `rounds ()` were instantiated each for encryption and decryption. These 13 instances were cascaded to each other and the last instance was cascaded to the `roundlast ()` module.

- **Module Prototype 6**: `roundlast ()`

```
module roundlast(input clk, input decrypt, input [127:0]data,
                 input [127:0]key, output reg DataIn)
```

Within this module, `AddRoundKey (), SubBytes (), ShiftRows (), InvAddRoundKey (), InvSubBytes ()` and `InvShiftRows` transformations were implemented. In this implementation, `SubBytes ()` and `ShiftRows ()`, and, `InvSubBytes ()` and `InvShiftRows ()` were implemented inside a single block and within in a single clock. If the input signal `decrypt = 0`, then this module would perform encryption, and if `decrypt = 1`, this module would perform decryption. Hence, the output of this module would be the encrypted data for input signal `decrypt = 0`, and, the output of this module would be the decrypted data for input signal `decrypt = 1`.

4.5 AES-256 Interface with MicroBlaze

The implemented ***AES-256*** module was packaged into a custom IP core using Vivado IP Packager tool. This ***AES-256*** IP core was connected to MicroBlaze using AXI Interconnect. In this interface, MicroBlaze was the master device and ***AES-256*** IP core was the slave device. The overall block diagram for interfacing ***AES-256*** IP core and MicroBlaze processor is shown in Figure 28 and the Vivado Block Design of this overall implementation is shown in Figure 53 (Appendix C). The resource utilization for this overall Vivado Block Design is shown in Figure 54 and Figure 55 in Appendix D.

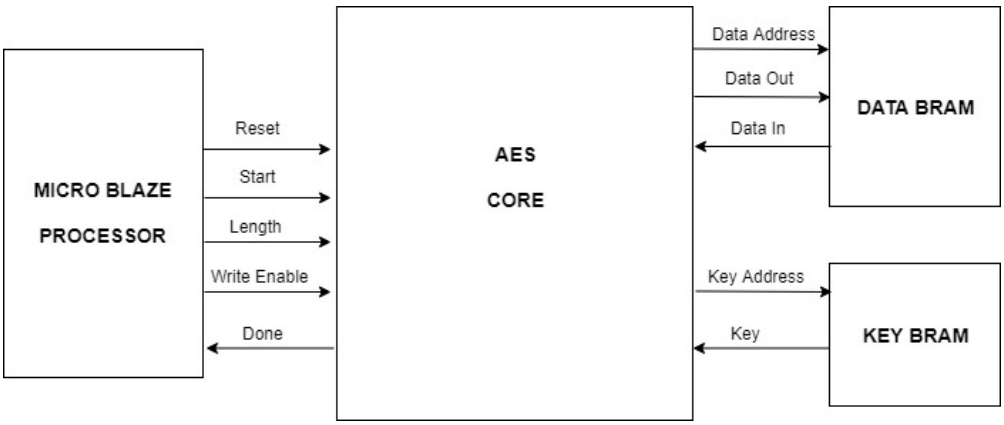


Figure 28: Block Diagram of AES-256 IP Core with MicroBlaze

4.6 Total On-Chip Power Consumption

The total on-chip power is also known as 'thermal power'. It was obtained using Equation (24) [24].

```
Total On-Chip Power = Static + Design Dynamic ......(24)
```

Where,

```
Static = Device Static + Design Static .............(25)
```

Device Static is the Transistor Leakage Power when the device is powered and not configured [24]. Design Static refers to the power when the device is configured and there is no switching activity [24]. It includes static power in I/O DCI terminations. Finally, Design Dynamic is the average power from user logic utilization and switching activity [24].

The power analysis of the implemented design was performed using the Vivado™ Power Analysis tool present in the Vivado Integrated Design Environment (IDE). It performs power estimation through all stages of the flow: after synthesis, after placement, and after routing. It is most accurate post-route since it can read from the implemented design database the exact logic and routing resources used [24]. A detailed on-chip power consumption values for the

implemented design is shown in Figure 56 in Appendix E. The Vivado™ Power Analysis tool also provides values for Junction Temperature (°C) and Effective Thermal Resistance to Air (ΘJA (°C/W)). Junction Temperature is temperature of the device in operation [24]. Effective Thermal Resistance to Air is also known as Theta-JA, and T_{JA} [24]. This coefficient defines how power is dissipated from the FPGA silicon to the environment (device junction to ambient air) [24]. The summary of power utilization for overall block design is shown in Figure 57 in Appendix E.

Chapter 5 describes how software implementation of ***PBKDF2*** based on ***HMAC-SHA1*** for ***WPA2-PSK*** device authentication, and hardware implementation of ***AES-256*** used in device confidentiality was tested. It goes through various test setups that were used and explains the results obtained from the tests. Finally, it compares the results of the implementation done in this thesis with previous existing implementations.

Chapter 5: Testing and Result

5.1 Overview

The goal of this thesis was efficient software implementation of ***PBKDF2*** based on ***HMAC-SHA1*** using C programming language, and, efficient hardware implementation of ***AES-256*** cipher using Verilog HDL. In the ***Wi-Fi*** communication, ***PBKDF2*** was used for device authentication, while ***AES-256*** was used for data confidentiality. To test the validity of the goals achieved by the implementations described in Chapter 3 and Chapter 4, test setup illustrated by the block diagram shown in Figure 29 was arranged. The setup has 3 main components:

- Nexys4 board with code for the implemented hardware and software design.
- MRF24WG0MA PMOD ***Wi-Fi***
- Laptop

For testing software implementation of ***PBKDF2*** based on ***HMAC-SHA1***, a single instance of block diagram shown in Figure 29 was used. Whereas, for testing hardware implementation of ***AES-256*** cipher, two instances of block diagram shown in Figure 29 were used. Additionally, a wireless access point was created using a mobile hotspot to complete the test setup for both the tests. The mobile hotspot and wireless access point will be used interchangeably in this chapter.

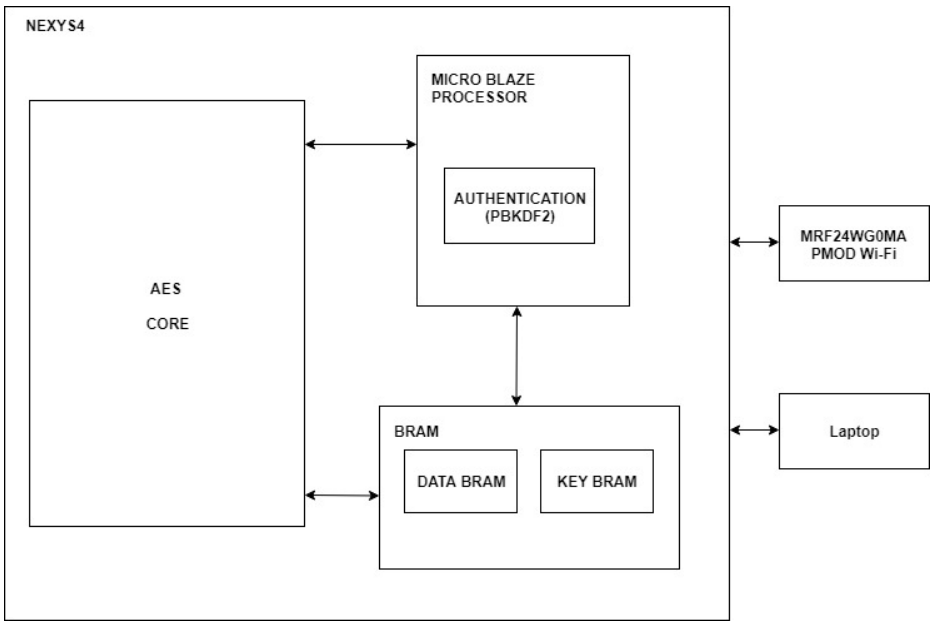


Figure 29: Overall Block Diagram for Testing

5.2 WPA2-PSK Testing and Result

The test for authentication of ***Wi-Fi*** using ***WPA2-PSK*** was performed with mobile hotspot as the access point. The access point was configured with "**TestWifi**" as the SSID and "**TestPassword**" as the Passphrase (Figure 30). The wireless module (MRF24WG0MA PMOD ***Wi-Fi***) interfaced with Nexys4 board (Figure 31) running the implemented design was used as the end node to connect to the access point. The Nexys4 board had USB-to-UART module running on it. The board was connected to a serial terminal software called **TERA TERM,** which opened a serial COM port with settings:

- Baud rate: 115200 bps
- Data size: 8- bit
- Parity: None
- Stop bits: 1-bit
- Flow Control: None

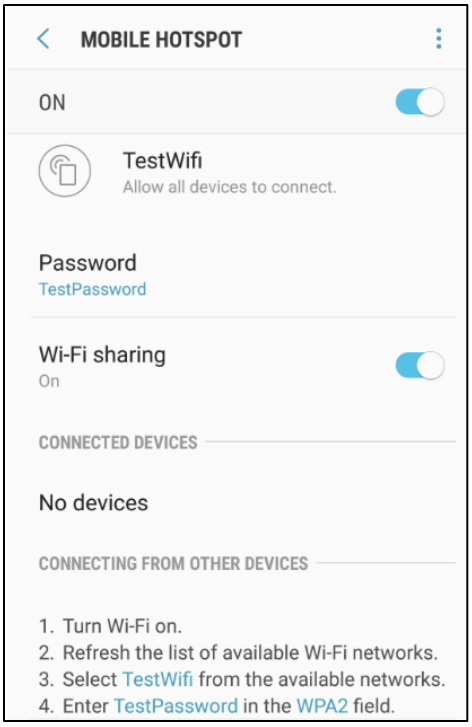


Figure 30: Mobile Hotspot Configuration

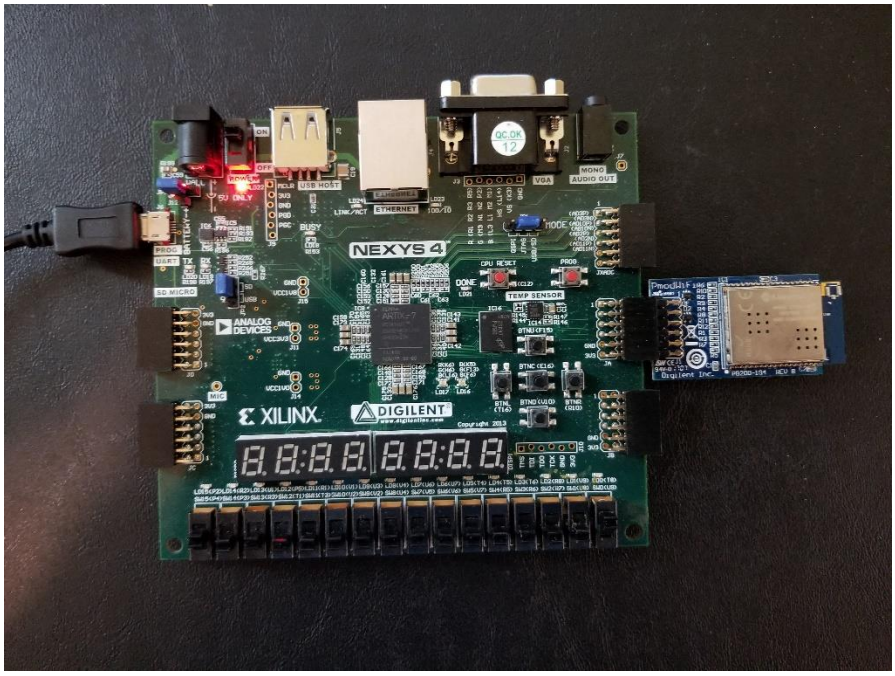


Figure 31: Nexys4 Board with MRF24WG0MA PMOD Wi-Fi

When this test setup connected to the serial term software was running, the values for SSID and Passphrase were prompted to the user on the terminal screen. The user would then type the values of SSID and Passphrase on the terminal. 4 test cases of SSID and Passphrase were used to check the validity of the authentication process. A table containing these 4 test cases along with the results of the tests is shown in Table 1.

Table 1: Table of Test Cases and Results for Authentication

CASE	RESULT
Incorrect SSID and Incorrect Passphrase	Authentication Failed
Correct SSID and Incorrect Passphrase	Authentication Failed
Incorrect SSID and Correct Passphrase	Authentication Failed
Correct SSID and Correct Passphrase	Authentication Successful

When incorrect values of SSID and/or Passphrase were entered in the end nodes using the terminal software, their authentication failed, and a communication channel was not created (Figure 32, Figure 33 and Figure 34). When SSID and Passphrase information of the access point were correctly entered in the end node, the access point was able to successfully authenticate it (Figure 35). After successful authentication, the access point was able to create a communication channel between itself and the end node.

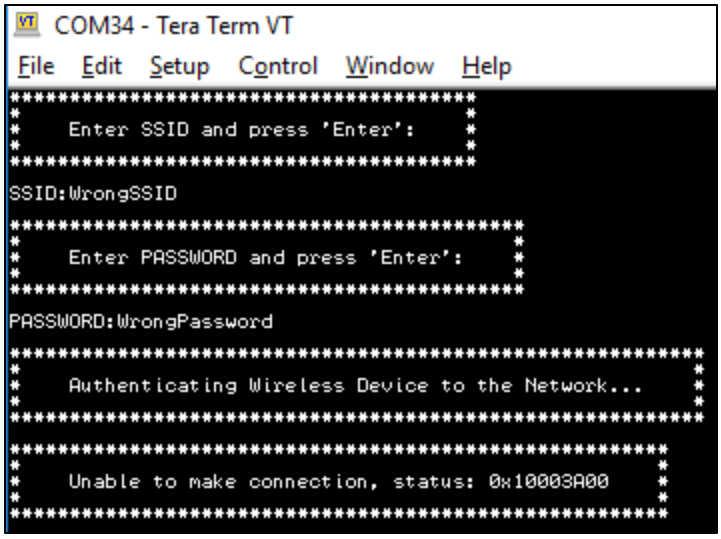


Figure 32: Failed Authentication with Incorrect SSID and Incorrect Password

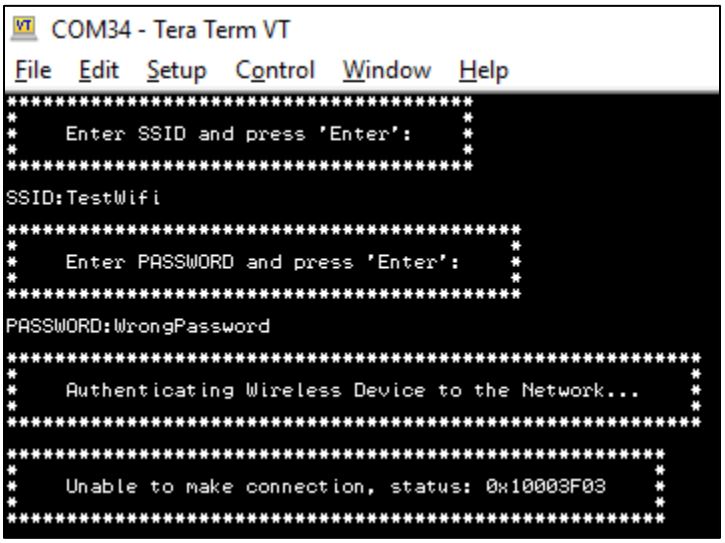


Figure 33: Failed Authentication with Correct SSID and Incorrect Password

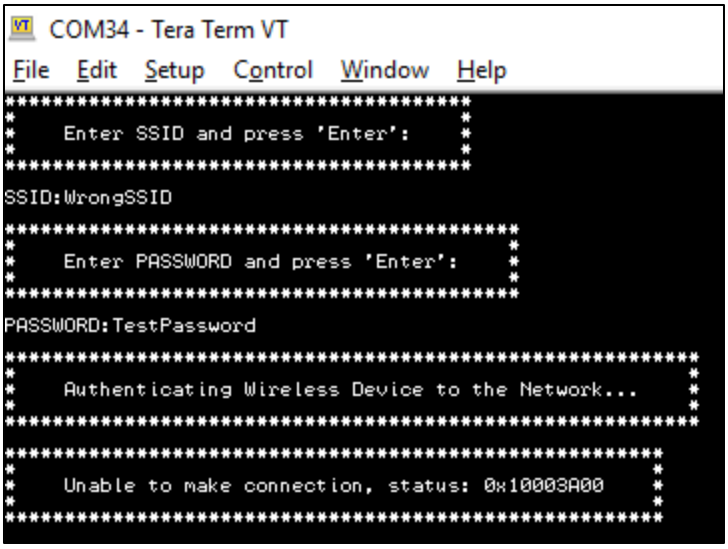


Figure 34: Failed Authentication with Incorrect SSID and Correct Password

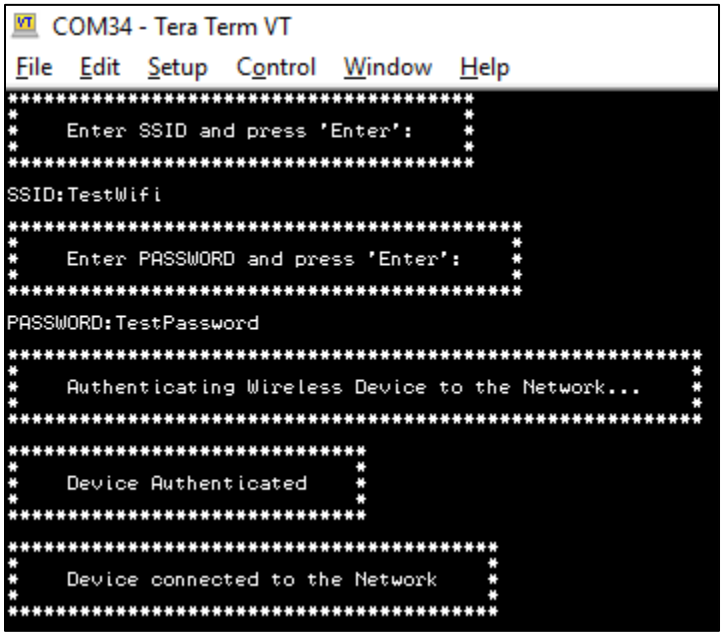


Figure 35: Successful Authentication with Correct SSID and Correct Password

5.3 WPA2-PSK Performance Evaluation

The performance of ***PBKDF2***, ***HMAC*** and ***SHA1*** operations using the implemented software described in Chapter 3 was compared with the existing design used in the MRF24WG0MA PMOD ***Wi-Fi*** software library. Latency (us) was used as the performance metric for all 3 operations**.** Latency was defined as the time (in us) required to complete the given operation.

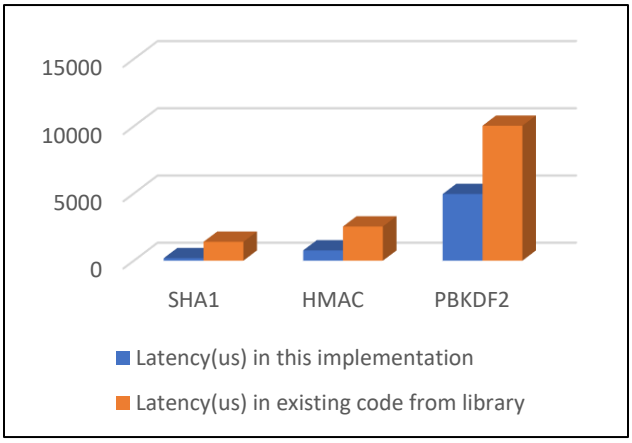
To determine the latency of an operation, initially, an AXI timer with a resolution of 10 ns was configured. Then, the following steps were performed:

- **Step1**: Timer was started.
- **Step2**: Timer value was read.
- **Step3**: The operation to be profiled was started.
- **Step4**: Timer was stopped after the completion of the function
- **Step5**: The Timer value was read.

These 5 steps were performed for ***SHA1***, ***HMAC*** and ***PBKDF2*** operations. The comparison of the latency results for these 3 operations is shown in Table 2.

Table 2: Latency Comparison of Functions used in WPA2-PSK Authentication

Operation	Latency(us) in this implementation	Latency(us) in existing code from library
SHA1	184.63	1404.27
HMAC	767. 36	2546.47
PBKDF2	4960000	10040000



In Table 2, the latency of 3 main operations: ***SHA1***, ***HMAC*** and ***PBKDF2*** were tabulated. In all the 3 operations, the values of the implemented design described in Chapter 3 was less compared to the values of the existing design from the MRF24WG0MA PMOD ***Wi-Fi*** software library. From these results, we can see that the implemented design was more efficient in terms of latency as compared to the design in MRF24WG0MA PMOD ***Wi-Fi*** software library. These results were attributed to the following reasons:

- In the implementation of ***SHA1*** operation in MRF24WG0MA PMOD ***Wi-Fi*** software library, **memcpy ()** and **memset ()** functions were used to copy every 512-bit input block of data into a temporary buffer. But, in the design described in Chapter 3, the need for the use of these functions were eliminated.
- In MRF24WG0MA PMOD ***Wi-Fi*** software library, nested functions were used to implement 80 rounds of ***SHA1*** operation per 512-bit input data block, Whereas, in the implementation described in Chapter 3, the same operation was implemented using loops within a single function. This decreased the overhead delay that was used to branch to multiple nested functions.
- In the implementation of ***HMAC*** operation in MRF24WG0MA PMOD ***Wi-Fi*** software library, the XOR operation of `opad` and `ipad` were done using 2 separate loops. But the same operation was done within a single loop for the design described in Chapter 3.
- In MRF24WG0MA PMOD ***Wi-Fi*** software library, the ***PBKDF2*** was implemented using nested functions. Whereas in the implementation described in Chapter 3, the same operation was implemented within a single function. This decreased the overhead delay that was used to branch to different functions.

5.4 AES-256 Testing and Result

The test for data confidentiality in ***Wi-Fi*** using ***AES-256*** was performed with mobile hotspot as the access point. This access point was configured with "**TestWifi**" as the SSID and "**TestPassword**" as the Passphrase (Figure 30). Two setups shown in Figure 31 were used as end nodes to connect to the access point. When SSID and Passphrase information of the access point were correctly entered in the two end nodes, the access point was able to successfully authenticate them (Figure 35). Data was communicated between these nodes using TCP protocol through the access point. The following two tests were conducted to verify data confidentiality using ***AES-256***:

5.4.1 TCP Server and TCP Client

In this test setup, one of the wireless end nodes was coded to run as a TCP server while the other node was coded to run as a TCP client. Both end nodes were authenticated when correct values of SSID and Passphrase were entered. After authentication, the TCP server opened a socket listening at address **192.168.43.7:80**. The TCP Client would then try to connect to the server. After a successful TCP connection, encrypted data was transferred from the client to the server (Figure 36).

Figure 36: Encrypted Data Sent by TCP Client to TCP Server

When the server received the encrypted data, it would decrypt it. In Figure 37, the server has received the encrypted data and successfully decrypted it to obtain the original data sent by the client.

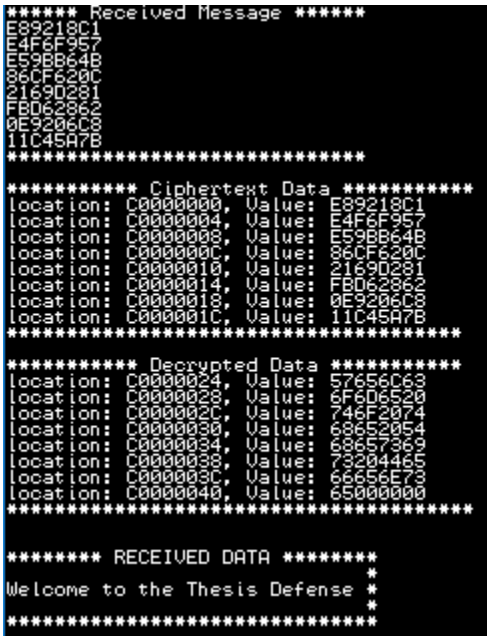


Figure 37: Decryption of Data Received by TCP Server from TCP Client

Now, the server would append additional data to decrypted data, encrypt this new data, and send this encrypted data back to the client (Figure 38).

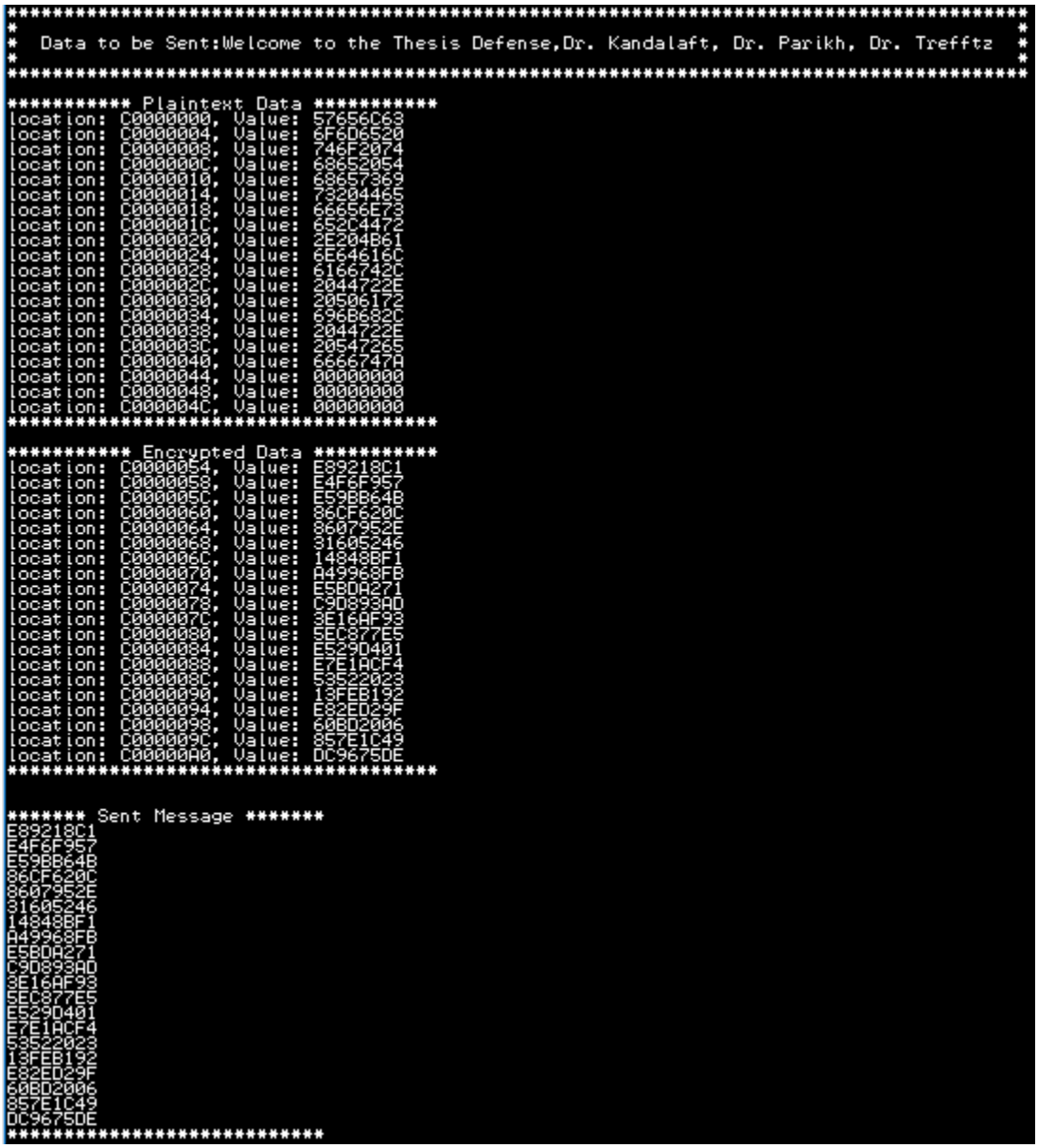


Figure 38: Encrypted Data Sent by TCP Server to TCP Client

When the client received the encrypted data, it would decrypt it. In Figure 39, the client has correctly received the encrypted data and successfully decrypted it to obtain the original data sent by the server.

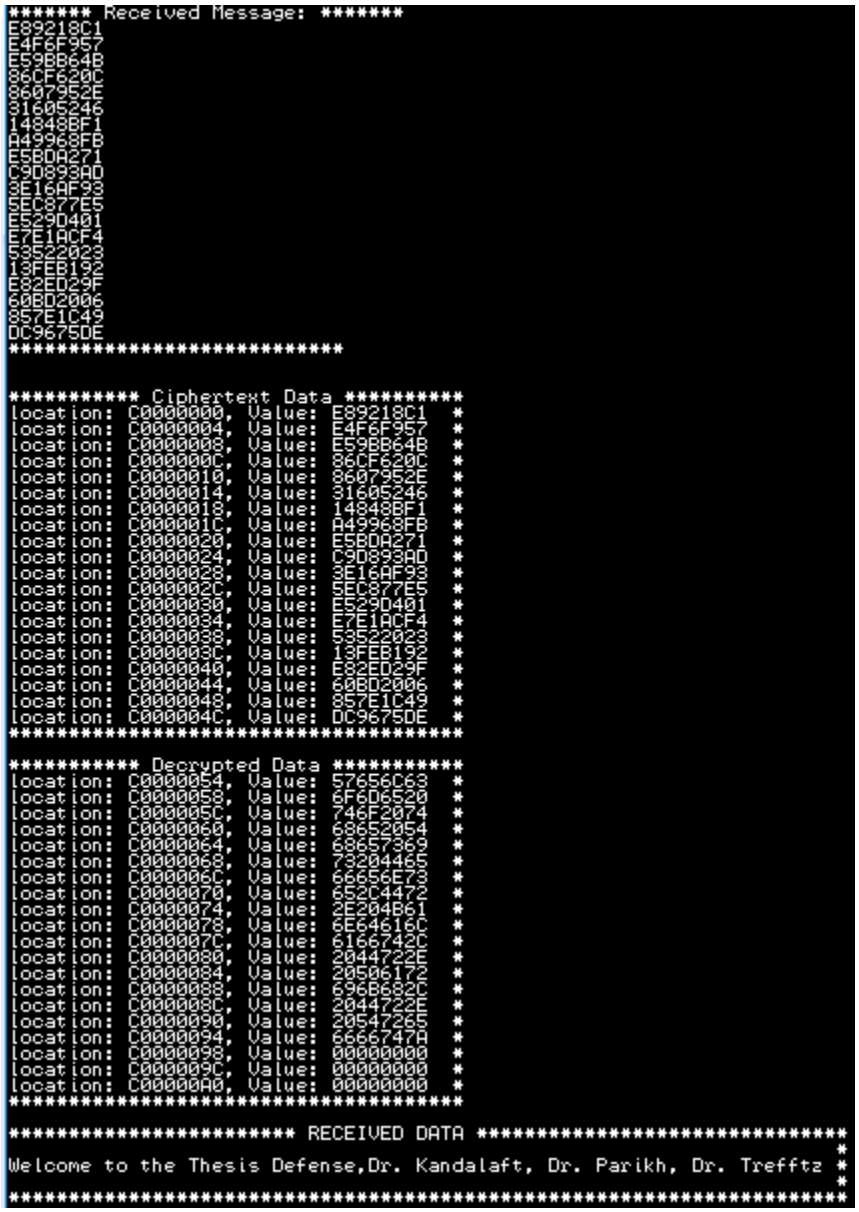


Figure 39: Decryption of Data Received by TCP Client from TCP Server

5.4.2 HTTP Server and Web Browser

In this test setup, one of the wireless end nodes was coded to run as a HTTP server which could serve HTTP GET requests. This server was able to serve two webpages with URL **192.168.43.7/aes** and **192.168.43.7/config**. At the beginning, this device was first authenticated by entering correct values of SSID and Passphrase. After authentication, the HTTP server opened a socket listening at address 192.168.43.7:80. Now, a test laptop was connected to the same access point and a web browser application was run on it. When the address **http://192.168.43.7/aes** was entered as the URL, the web browser would show the webpage seen in Figure 40.

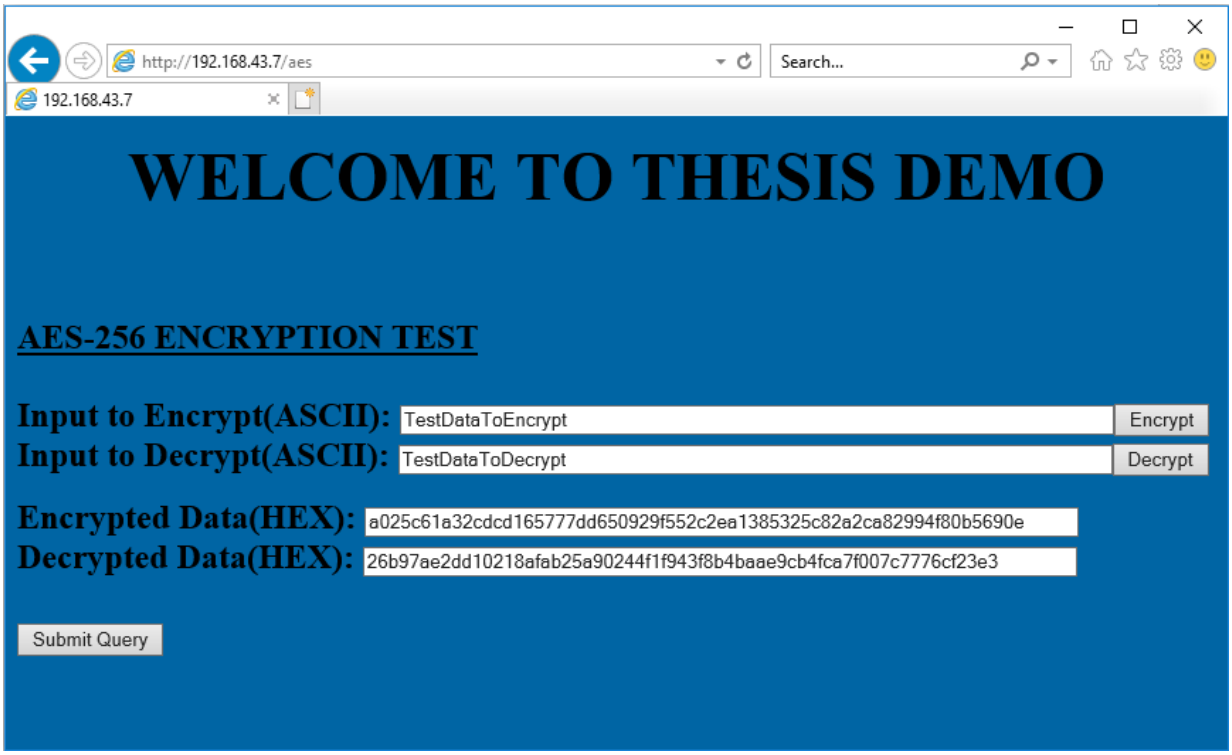


Figure 40: AES Webpage hosted by HTTP server

In the webpage shown in Figure 40, **Input to Encrypt (ASCII)** field and **Input to Decrypt (ASCII)** field were filled with test data. When the **Encrypt** and **Decrypt** buttons were

pressed in the webpage, the **Encrypted Data (HEX)** and **Decrypted Data (HEX)** fields were filled respectively. Finally, when the **Submit Query** button was pressed, the data in the **Encrypted Data (HEX)** field and **Decrypted Data (HEX)** field were transferred to the HTTP server using HTTP protocol.

In Figure 41, the encrypted and decrypted set of data received by the HTTP server that was sent from the web browser is shown. The results of the decryption of the encrypted data, and encryption of decrypted data is also shown in Figure 41. It can be observed that the original data entered in **Input to Encrypt (ASCII)** field and **Input to Decrypt (ASCII)** field from webpage in Figure 40 matches with the decrypted and encrypted values from Figure 41.

Figure 41:Decryption and Encryption of Data received by HTTP server

5.5 AES-256 Performance Evaluation

With reference to Figure 27 in Chapter 4, two different logics were written for `AddRoundKey ()`, `SubBytes ()`, `ShiftRows ()`, `MixColumns ()`, `InvAddRoundKey ()`, `InvSubBytes ()`, `InvShiftRows ()` and `InvMixColumns ()` transformations in AES-***256*** core. The first logic used the Look Up Table (LUT) method, whereas the second logic used the BRAM method. In the LUT method, the transformations were performed asynchronously, while in BRAM method, the transformations were performed synchronously. The resource utilization for LUT method is shown in Figure 42 and Figure 43, and resource utilization for BRAM method is shown in Figure 44 and Figure 45.

Resource	Utilization	Available	Utilization...
LUT	23382	63400	36.88
FF	10319	126800	8.14
BRAM	1	135	0.74
IO	52	210	24.76
BUFG	1	32	3.13

Figure 42: Resources Utilization Table for LUT Implementation

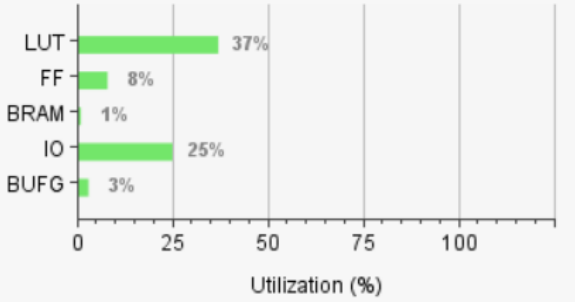


Figure 43: Graph for Resource Utilization for LUT Implementation

Resource	Utilization	Available	Utilization...
LUT	7699	63400	12.14
FF	8562	126800	6.75
BRAM	113	135	83.70
IO	52	210	24.76
BUFG	1	32	3.13

Figure 44: Resources Utilization Table for BRAM Implementation

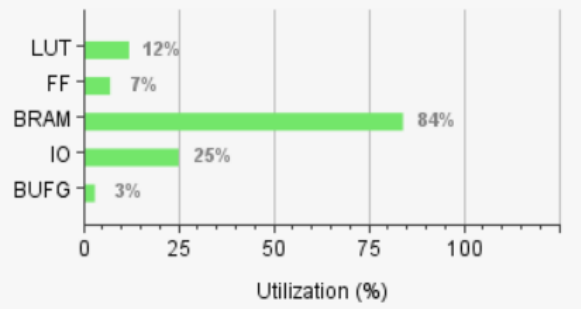


Figure 45: Graph for Resource Utilization for BRAM Implementation

From Figures 42, 43, 44 and 45, we see that **LUT** and **FF** utilization is greater in the LUT method as compared to the BRAM method. Whereas, the **BRAM** utilization is less in LUT method in comparison to the BRAM method. The **IO** and **BUFG** utilization are same in both cases. The **LUT** utilization is greater in LUT method because, when transformations are implemented asynchronously, the memory related to the transformation in the design will be inferred as a Lookup table. Whereas, the **BRAM** utilization is greater in **BRAM** method because, when transformations are implemented synchronously, the memory related to the transformation in the design will be inferred as a Block RAM. Hence, there is a trade-off between utilization of **LUTs** and **BRAMs** in the two designs.

In this thesis, the LUT method of ***AES-256*** was arbitrarily selected to be converted to the ***AES-256*** IP core and be interfaced with MicroBlaze Processor. The performance of this

implementation was compared with the other existing implementations described in [25], [26], [27], [28]. For the purpose of comparison in the following sections in Chapter 5 and Chapter 6, the implemented design of AES-256 using LUT is referred as Design 1, and implementations referenced from [25], [26], [27], [28] are referred as Design 2, Design 3, Design 4 and Design 5.

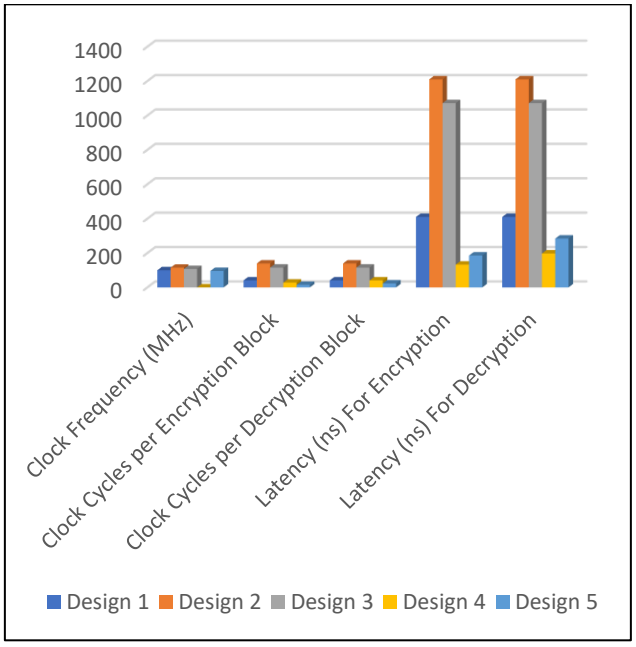
5.5.1 Latency Comparison

Latency is the time taken by the implemented design to produce one block of 128-bit output data for one block of 128-bit input data. This value depends on the number of clock cycles per encrypted/decrypted block and the frequency of the clock used. Latency was manually calculated using Equation (26) and the values for all five design are tabulated in Table 3.

```
Latency = Clock Cycles per Output Block x Clock Frequency .....(26)
```

Table 3: Comparison of Implemented Designs Based on Latency

Design	Clock Frequency (MHz)	Clock Cycles per Encryption/ Decryption Block	Latency (ns) For Encryption/ Decryption Block
Design 1	100	41/ 41	410/ 410
Design 2	114.877	139/ 139	1209.981/ 1209.981
Design 3	107.3	115/ 115	1071.761/ 1071.761
Design 4	-	28/ 41	132.5/ 197.5
Design 5	95.721	15/ 23	185.52/ 284.264



In Table 3, it was seen that Design 1 had less latency compared to Designs 2 and 3, while it wasn't able to match or better the latency from Designs 4 and 5. This was because in Designs 1,4 and 5, the multiplication tables for `MixColumns()` and `InvMixColumns()` operations were implemented using Look Up Tables, as opposed to Designs 2 and 3, which used actual multiplication calculations over Galois Field. Performing actual multiplication calculations would take more clock cycles to complete than just extracting the values from Look Up Tables. Even though Design 1 and 5 both used Look Up Tables, Design 5 had a less latency because it used pipelined architecture.

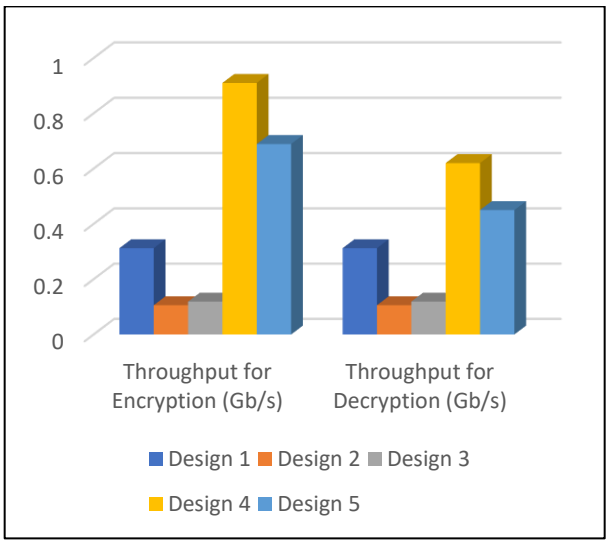
5.5.2 Throughput Comparison

Throughput was manually calculated using equation (27) and the values for all five designs are tabulated in Table 4. In equation (27), the block size is 128-bit.

```
Throughput = Block Size / Latency ..................................... (27)
```

Table 4: Comparison of Implemented Designs Based on Throughput

Design	Throughput for Encryption (Gb/s)	Throughput for Decryption (Gb/s)
Design 1	0.312	0.312
Design 2	0.106	0.106
Design 3	0.119	0.119
Design 4	0.91	0.62
Design 5	0.689	0.45



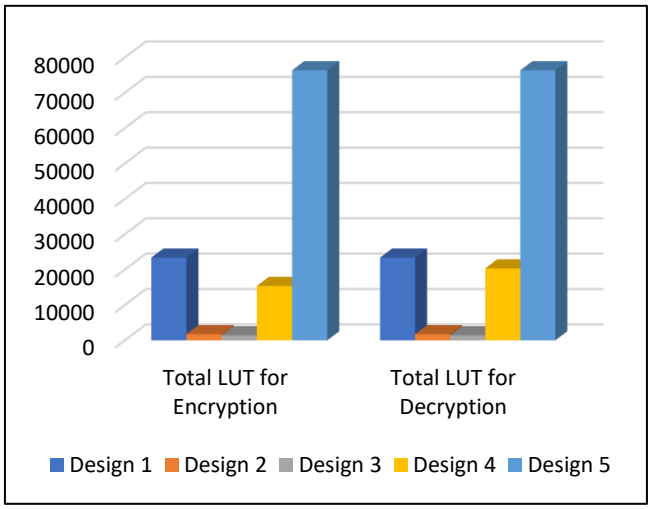
In Table 4, it was observed that Design 1 had higher throughput as compared to Design 2 and 3 but had a lower throughput in comparison to Design 4 and 5. From Equation (27), it was observed that throughput is related to latency and block size. Since, block size is same for all 5 designs, the only variable affecting throughput between the five designs is latency. Hence, latency and throughout follow the same result pattern for the five designs.

5.5.3 Resource Utilization Comparison

The LUT slice utilization for the five implemented designs are shown in Table 5.

Table 5: Comparison of Implemented Designs Based on Resource Utilization

Design	Total LUT for Encryption	Total LUT for Decryption
Design 1	23382	23382
Design 2	1737	1737
Design 3	1428	1428
Design 4	15376	20324
Design 5	76365	76365



In Table 5, it was observed that Design 1 had less LUT slice utilization as compared to Design 5, but, had a higher LUT slice utilization in comparison to Designs 2, 3 and 4. This is because, in Design 1, `MixColumns ()` and `InvMixColumns ()` operations were implemented using Look Up Tables, which would increase the LUT resource utilization. Even though Design 1 and 5 both used Look Up Tables, Design 5 had a higher LUT resource utilization because it used pipelined architecture.

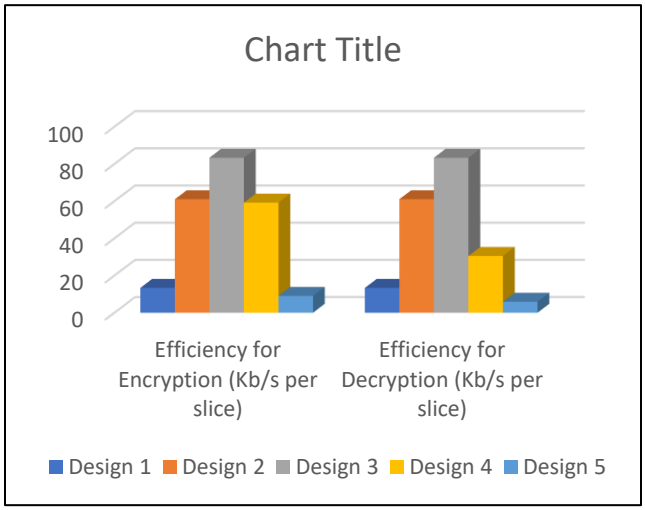
5.5.4 Efficiency Comparison

Efficiency was manually calculated using equation (28) and the values for all the five designs are shown in Table 6.

```
Efficiency = Throughput / Number of LUT slices ..................... (28)
```

Table 6: Comparison of Implemented Designs Based on Efficiency

Design	Efficiency for Encryption (Kb/s per slice)	Efficiency for Decryption (Kb/s per slice)
Design 1	13.344	13.344
Design 2	61.025	61.025
Design 3	83.333	83.333
Design 4	59.183	30.505
Design 5	9.022	5.892



In Table 6, it was seen that Design 1 had better efficiency as compared to Design 5, but, had a lower efficiency when compared to Designs 2,3 and 4. This was because when compared to Design 1, the ratio of throughput to number of LUT slices was less for Design 5, while this ratio was higher for Design 2,3 and 4.

5.5.5 Summary

From the results of different performance metrics seen in Tables 3,4,5 and 6, it can be inferred that there is a tradeoff between latency and LUT resource utilization. These two metrics seem to be inversely proportional to each other as shown in Figure 46 and Figure 47.

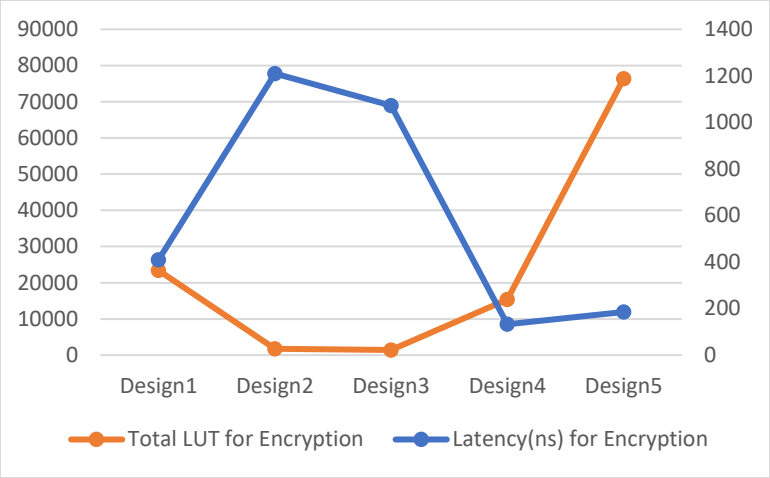


Figure 46: Comparison between Latency and LUT for Encryption in all Designs

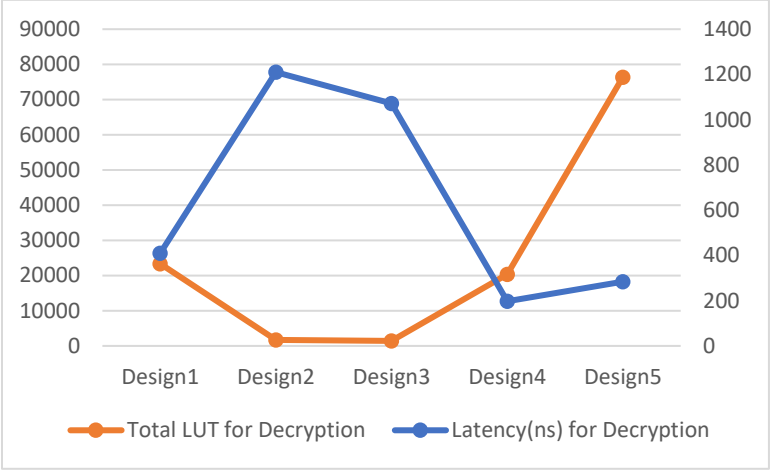


Figure 47: Comparison between Latency and LUT for Decryption in all Designs

From Figure 46 and Figure 47, we observe that in all designs as the number of LUTs increases the latency decreases and vice versa. Hence, an efficient design would contain a compromise between latency and LUT resource utilization values. If we observe the formula to calculate latency in equation (26), we see that by using a larger clock frequency we can achieve lower latency values for the same number of clock cycles. Hence, using a FPGA board with a higher clock source could be a method to achieve better latency without increasing the LUT

resource utilization. This in turn will improve the throughput and efficiency results as well. Another way to improve the efficiency would be to reduce the LUT resource utilization by using most of the Block RAM(BRAM) available in FPGA chip. FPGA chips have limited BRAMs in comparison to Look up table. By selecting a FPGA chip supporting large amount of BRAMs and writing Verilog code in a manner that the memory locations infer BRAM instead of LUT, we can reduce the LUT resource utilization without increasing the latency. Hence, this will improve the overall efficiency of the design.

Chapter 6 reiterates the outcome of the implemented design described in Chapters 3 and 4. It also discusses conclusions that can be drawn from the results of the various tests performed in Chapter 5.

Chapter 6: Conclusion

This thesis involved software implementation of ***PBKDF2*** based on ***HMAC-SHA1*** using C programming language, and hardware implementation of ***AES-256*** using Verilog HDL. ***PBKDF2*** was used for device authentication while ***AES-256*** was used for data confidentiality in ***WPA2-PSK***. The overall implementation was designed and tested on Nexys4 FPGA board. The performance of this implementation was compared with other existing designs.

The latency(us) of software implementation of ***SHA1***, ***HMAC*** and ***PBKDF2*** operations were compared to the latency of the same operations obtained from MRF24WG0MA PMOD ***Wi-Fi*** software library. The latency of these operations from the implemented design was less when compared to that from MRF24WG0MA PMOD ***Wi-Fi*** software library.

Latency (ns), throughput (Gb/s), resource utilization (Number of Slices) and efficiency (Kb/s per slice) were the performance metrics used for hardware implementation of ***AES-256***. These metrics were compared for the results from Design 2,3,4 and 5. Design 1 had a better latency and throughput results in comparison to Design 2 and 3, while it could not better the latency and throughput results when compared with Design 4 and 5. Similarly, Design 1 had a better LUT slice utilization and efficiency results when compared to Design 5 but could not better the results when compared to Designs 2,3 and 4.

From these results, it can be concluded that latency and LUT resource utilization were inversely proportional to each other. Hence, an efficient design would be a compromise between latency and LUT resource utilization values.

Chapter 7 discusses the limitations of the implemented design described in Chapters 3 and 4. It also elaborates on possible future work to overcome these limitations.

Chapter 7: Future Work

This thesis involved software implementation of ***PBKDF2*** based on ***HMAC-SHA1*** using C programming language for device authentication, and hardware implementation of ***AES-256*** using Verilog HDL for data confidentiality in ***WPA2-PSK***. There are several areas of potential future improvements for device authentication and data confidentiality in ***WPA2-PSK***. These are listed below:

- Though an efficient design in terms of latency was obtained from software implementation of ***PBKDF2***, ***HMAC*** and ***SHA1***, a better implementation would be one on hardware using Verilog HDL.
- At the end of ***PBKDF2*** operation, a 256-bit PMK was obtained. For future work, the compete authentication process involving exchange of PMK between ***Wi-Fi*** device and access point using 4-way handshake could be implemented.
- The ***Wi-Fi*** module (MRF24WG0MA) used in the thesis already had an AES core linked to it that had to be used. Hence, the implemented ***AES-256*** module from this thesis was used along with the existing AES core from the MRF24WG0MA ***Wi-Fi*** module. In other words, an extra layer of encryption using the implemented ***AES-256*** was added over the existing AES encryption layer from MRF24WG0MA PMOD ***Wi-Fi*** library. For future development, it would be better to use a ***Wi-Fi*** module that can be directly interfaced with the implemented ***AES-256***, thus, it would only use a single layer of encryption.

Appendices

A. AES Lookup Tables

Table 7: S-Box Lookup Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 8: Inverse S-Box Lookup Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Table 9: Mul2 Lookup Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	02	04	06	08	0a	0c	0e	10	12	14	16	18	1a	1c	1e
1	20	22	24	26	28	2a	2c	2e	30	32	34	36	38	3a	3c	3e
2	40	42	44	46	48	4a	4c	4e	50	52	54	56	58	5a	5c	5e
3	60	62	64	66	68	6a	6c	6e	70	72	74	76	78	7a	7c	7e
4	80	82	84	86	88	8a	8c	8e	90	92	94	96	98	9a	9c	9e
5	a0	a2	a4	a6	a8	aa	ac	ae	b0	b2	b4	b6	b8	ba	bc	be
6	c0	c2	c4	c6	c8	ca	cc	ce	d0	d2	d4	d6	d8	da	dc	de
7	e0	e2	e4	e6	e8	ea	ec	ee	f0	f2	f4	f6	f8	fa	fc	fe
8	1b	19	1f	1d	13	11	17	15	0b	09	0f	0d	03	01	07	05
9	3b	39	3f	3d	33	31	37	35	2b	29	2f	2d	23	21	27	25
A	5b	59	5f	5d	53	51	57	55	4b	49	4f	4d	43	41	47	45
B	7b	79	7f	7d	73	71	77	75	6b	69	6f	6d	63	61	67	65
C	9b	99	9f	9d	93	91	97	95	8b	89	8f	8d	83	81	87	85
D	bb	b9	bf	bd	b3	b1	b7	b5	ab	a9	af	ad	a3	a1	a7	a5
E	db	d9	df	dd	d3	d1	d7	d5	cb	c9	cf	cd	c3	c1	c7	c5
F	fb	f9	ff	fd	f3	f1	f7	f5	eb	e9	ef	ed	e3	e1	e7	e5

Table 10: Mul3 Lookup Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	03	06	05	0c	0f	0a	09	18	1b	1e	1d	14	17	12	11
1	30	33	36	35	3c	3f	3a	39	28	2b	2e	2d	24	27	22	21
2	60	63	66	65	6c	6f	6a	69	78	7b	7e	7d	74	77	72	71
3	50	53	56	55	5c	5f	5a	59	48	4b	4e	4d	44	47	42	41
4	c0	c3	c6	c5	cc	cf	ca	c9	d8	db	de	dd	d4	d7	d2	d1
5	f0	f3	f6	f5	fc	ff	fa	f9	e8	eb	ee	ed	e4	e7	e2	e1
6	a0	a3	a6	a5	ac	af	aa	a9	b8	bb	be	bd	b4	b7	b2	b1
7	90	93	96	95	9c	9f	9a	99	88	8b	8e	8d	84	87	82	81
8	9b	98	9d	9e	97	94	91	92	83	80	85	86	8f	8c	89	8a
9	ab	a8	ad	ae	a7	a4	a1	a2	b3	b0	b5	b6	bf	bc	b9	ba
A	fb	f8	fd	fe	f7	f4	f1	f2	e3	e0	e5	e6	ef	ec	e9	ea
B	cb	c8	cd	ce	c7	c4	c1	c2	d3	d0	d5	d6	df	dc	d9	da
C	5b	58	5d	5e	57	54	51	52	43	40	45	46	4f	4c	49	4a
D	6b	68	6d	6e	67	64	61	62	73	70	75	76	7f	7c	79	7a
E	3b	38	3d	3e	37	34	31	32	23	20	25	26	2f	2c	29	2a
F	0b	08	0d	0e	07	04	01	02	13	10	15	16	1f	1c	19	1a

Table 11: Mul9 Lookup Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	09	12	1b	24	2d	36	3f	48	41	5a	53	6c	65	7e	77
1	90	99	82	8b	b4	bd	a6	af	d8	d1	ca	c3	fc	f5	ee	e7
2	3b	32	29	20	1f	16	0d	04	73	7a	61	68	57	5e	45	4c
3	ab	a2	b9	b0	8f	86	9d	94	e3	ea	f1	f8	c7	ce	d5	dc
4	76	7f	64	6d	52	5b	40	49	3e	37	2c	25	1a	13	08	01
5	e6	ef	f4	fd	c2	cb	d0	d9	ae	a7	bc	b5	8a	83	98	91
6	4d	44	5f	56	69	60	7b	72	05	0c	17	1e	21	28	33	3a
7	dd	d4	cf	c6	f9	f0	eb	e2	95	9c	87	8e	b1	b8	a3	aa
8	ec	e5	fe	f7	c8	c1	da	d3	a4	ad	b6	bf	80	89	92	9b
9	7c	75	6e	67	58	51	4a	43	34	3d	26	2f	10	19	02	0b
A	d7	de	c5	cc	f3	fa	e1	e8	9f	96	8d	84	bb	b2	a9	a0
B	47	4e	55	5c	63	6a	71	78	0f	06	1d	14	2b	22	39	30
C	9a	93	88	81	be	b7	ac	a5	d2	db	c0	c9	f6	ff	e4	ed
D	0a	03	18	11	2e	27	3c	35	42	4b	50	59	66	6f	74	7d
E	a1	a8	b3	ba	85	8c	97	9e	e9	e0	fb	f2	cd	c4	df	d6
F	31	38	23	2a	15	1c	07	0e	79	70	6b	62	5d	54	4f	46

Table 12: Mul11 Lookup Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	0b	16	1d	2c	27	3a	31	58	53	4e	45	74	7f	62	69
1	b0	bb	a6	ad	9c	97	8a	81	e8	e3	fe	f5	c4	cf	d2	d9
2	7b	70	6d	66	57	5c	41	4a	23	28	35	3e	0f	04	19	12
3	cb	c0	dd	d6	e7	ec	f1	fa	93	98	85	8e	bf	b4	a9	a2
4	f6	fd	e0	eb	da	d1	cc	c7	ae	a5	b8	b3	82	89	94	9f
5	46	4d	50	5b	6a	61	7c	77	1e	15	08	03	32	39	24	2f
6	8d	86	9b	90	a1	aa	b7	bc	d5	de	c3	c8	f9	f2	ef	e4
7	3d	36	2b	20	11	1a	07	0c	65	6e	73	78	49	42	5f	54
8	f7	fc	e1	ea	db	d0	cd	c6	af	a4	b9	b2	83	88	95	9e
9	47	4c	51	5a	6b	60	7d	76	1f	14	09	02	33	38	25	2e
A	8c	87	9a	91	a0	ab	b6	bd	d4	df	c2	c9	f8	f3	ee	e5
B	3c	37	2a	21	10	1b	06	0d	64	6f	72	79	48	43	5e	55
C	01	0a	17	1c	2d	26	3b	30	59	52	4f	44	75	7e	63	68
D	b1	ba	a7	ac	9d	96	8b	80	e9	e2	ff	f4	c5	ce	d3	d8
E	7a	71	6c	67	56	5d	40	4b	22	29	34	3f	0e	05	18	13
F	ca	c1	dc	d7	e6	ed	f0	fb	92	99	84	8f	be	b5	a8	a3

Table 13: Mul13 Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	0d	1a	17	34	39	2e	23	68	65	72	7f	5c	51	46	4b
1	d0	dd	ca	c7	e4	e9	fe	f3	b8	b5	a2	af	8c	81	96	9b
2	bb	b6	a1	ac	8f	82	95	98	d3	de	c9	c4	e7	ea	fd	f0
3	6b	66	71	7c	5f	52	45	48	03	0e	19	14	37	3a	2d	20
4	6d	60	77	7a	59	54	43	4e	05	08	1f	12	31	3c	2b	26
5	bd	b0	a7	aa	89	84	93	9e	d5	d8	cf	c2	e1	ec	fb	f6
6	d6	db	cc	c1	e2	ef	f8	f5	be	b3	a4	a9	8a	87	90	9d
7	06	0b	1c	11	32	3f	28	25	6e	63	74	79	5a	57	40	4d
8	da	d7	c0	cd	ee	e3	f4	f9	b2	bf	a8	a5	86	8b	9c	91
9	0a	07	10	1d	3e	33	24	29	62	6f	78	75	56	5b	4c	41
A	61	6c	7b	76	55	58	4f	42	09	04	13	1e	3d	30	27	2a
B	b1	bc	ab	a6	85	88	9f	92	d9	d4	c3	ce	ed	e0	f7	fa
C	b7	ba	ad	a0	83	8e	99	94	df	d2	c5	c8	eb	e6	f1	fc
D	67	6a	7d	70	53	5e	49	44	0f	02	15	18	3b	36	21	2c
E	0c	01	16	1b	38	35	22	2f	64	69	7e	73	50	5d	4a	47
F	dc	d1	c6	cb	e8	e5	f2	ff	b4	b9	ae	a3	80	8d	9a	97

Table 14: Mul14 Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	0e	1c	12	38	36	24	2a	70	7e	6c	62	48	46	54	5a
1	e0	ee	fc	f2	d8	d6	c4	ca	90	9e	8c	82	a8	a6	b4	ba
2	db	d5	c7	c9	e3	ed	ff	f1	ab	a5	b7	b9	93	9d	8f	81
3	3b	35	27	29	03	0d	1f	11	4b	45	57	59	73	7d	6f	61
4	ad	a3	b1	bf	95	9b	89	87	dd	d3	c1	cf	e5	eb	f9	f7
5	4d	43	51	5f	75	7b	69	67	3d	33	21	2f	05	0b	19	17
6	76	78	6a	64	4e	40	52	5c	06	08	1a	14	3e	30	22	2c
7	96	98	8a	84	ae	a0	b2	bc	e6	e8	fa	f4	de	d0	c2	cc
8	41	4f	5d	53	79	77	65	6b	31	3f	2d	23	09	07	15	1b
9	a1	af	bd	b3	99	97	85	8b	d1	df	cd	c3	e9	e7	f5	fb
A	9a	94	86	88	a2	ac	be	b0	ea	e4	f6	f8	d2	dc	ce	c0
B	7a	74	66	68	42	4c	5e	50	0a	04	16	18	32	3c	2e	20
C	ec	e2	f0	fe	d4	da	c8	c6	9c	92	80	8e	a4	aa	b8	b6
D	0c	02	10	1e	34	3a	28	26	7c	72	60	6e	44	4a	58	56
E	37	39	2b	25	0f	01	13	1d	47	49	5b	55	7f	71	63	6d
F	d7	d9	cb	c5	ef	e1	f3	fd	a7	a9	bb	b5	9f	91	83	8d

B. Code Snippets

```
static const uint8_t sbox[256] = {
0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
};
```

Figure 48: Implementation of S-Box in C

```
int mul9[256]= {
0x00,0x09,0x12,0x1b,0x24,0x2d,0x36,0x3f,0x48,0x41,0x5a,0x53,0x6c,0x65,0x7e,0x77,
0x90,0x99,0x82,0x8b,0xb4,0xbd,0xa6,0xaf,0xd8,0xd1,0xca,0xc3,0xfc,0xf5,0xee,0xe7,
0x3b,0x32,0x29,0x20,0x1f,0x16,0x0d,0x04,0x73,0x7a,0x61,0x68,0x57,0x5e,0x45,0x4c,
0xab,0xa2,0xb9,0xb0,0x8f,0x86,0x9d,0x94,0xe3,0xea,0xf1,0xf8,0xc7,0xce,0xd5,0xdc,
0x76,0x7f,0x64,0x6d,0x52,0x5b,0x40,0x49,0x3e,0x37,0x2c,0x25,0x1a,0x13,0x08,0x01,
0xe6,0xef,0xf4,0xfd,0xc2,0xcb,0xd0,0xd9,0xae,0xa7,0xbc,0xb5,0x8a,0x83,0x98,0x91,
0x4d,0x44,0x5f,0x56,0x69,0x60,0x7b,0x72,0x05,0x0c,0x17,0x1e,0x21,0x28,0x33,0x3a,
0xdd,0xd4,0xcf,0xc6,0xf9,0xf0,0xeb,0xe2,0x95,0x9c,0x87,0x8e,0xb1,0xb8,0xa3,0xaa,
0xec,0xe5,0xfe,0xf7,0xc8,0xc1,0xda,0xd3,0xa4,0xad,0xb6,0xbf,0x80,0x89,0x92,0x9b,
0x7c,0x75,0x6e,0x67,0x58,0x51,0x4a,0x43,0x34,0x3d,0x26,0x2f,0x10,0x19,0x02,0x0b,
0xd7,0xde,0xc5,0xcc,0xf3,0xfa,0xe1,0xe8,0x9f,0x96,0x8d,0x84,0xbb,0xb2,0xa9,0xa0,
0x47,0x4e,0x55,0x5c,0x63,0x6a,0x71,0x78,0x0f,0x06,0x1d,0x14,0x2b,0x22,0x39,0x30,
0x9a,0x93,0x88,0x81,0xbe,0xb7,0xac,0xa5,0xd2,0xdb,0xc0,0xc9,0xf6,0xff,0xe4,0xed,
0x0a,0x03,0x18,0x11,0x2e,0x27,0x3c,0x35,0x42,0x4b,0x50,0x59,0x66,0x6f,0x74,0x7d,
0xa1,0xa8,0xb3,0xba,0x85,0x8c,0x97,0x9e,0xe9,0xe0,0xfb,0xf2,0xcd,0xc4,0xdf,0xd6,
0x31,0x38,0x23,0x2a,0x15,0x1c,0x07,0x0e,0x79,0x70,0x6b,0x62,0x5d,0x54,0x4f,0x46
};
```

Figure 49: Implementation of Mul9 Lookup Table in C

```
int mul11[256]={
0x00,0x0b,0x16,0x1d,0x2c,0x27,0x3a,0x31,0x58,0x53,0x4e,0x45,0x74,0x7f,0x62,0x69,
0xb0,0xbb,0xa6,0xad,0x9c,0x97,0x8a,0x81,0xe8,0xe3,0xfe,0xf5,0xc4,0xcf,0xd2,0xd9,
0x7b,0x70,0x6d,0x66,0x57,0x5c,0x41,0x4a,0x23,0x28,0x35,0x3e,0x0f,0x04,0x19,0x12,
0xcb,0xc0,0xdd,0xd6,0xe7,0xec,0xf1,0xfa,0x93,0x98,0x85,0x8e,0xbf,0xb4,0xa9,0xa2,
0xf6,0xfd,0xe0,0xeb,0xda,0xd1,0xcc,0xc7,0xae,0xa5,0xb8,0xb3,0x82,0x89,0x94,0x9f,
0x46,0x4d,0x50,0x5b,0x6a,0x61,0x7c,0x77,0x1e,0x15,0x08,0x03,0x32,0x39,0x24,0x2f,
0x8d,0x86,0x9b,0x90,0xa1,0xaa,0xb7,0xbc,0xd5,0xde,0xc3,0xc8,0xf9,0xf2,0xef,0xe4,
0x3d,0x36,0x2b,0x20,0x11,0x1a,0x07,0x0c,0x65,0x6e,0x73,0x78,0x49,0x42,0x5f,0x54,
0xf7,0xfc,0xe1,0xea,0xdb,0xd0,0xcd,0xc6,0xaf,0xa4,0xb9,0xb2,0x83,0x88,0x95,0x9e,
0x47,0x4c,0x51,0x5a,0x6b,0x60,0x7d,0x76,0x1f,0x14,0x09,0x02,0x33,0x38,0x25,0x2e,
0x8c,0x87,0x9a,0x91,0xa0,0xab,0xb6,0xbd,0xd4,0xdf,0xc2,0xc9,0xf8,0xf3,0xee,0xe5,
0x3c,0x37,0x2a,0x21,0x10,0x1b,0x06,0x0d,0x64,0x6f,0x72,0x79,0x48,0x43,0x5e,0x55,
0x01,0x0a,0x17,0x1c,0x2d,0x26,0x3b,0x30,0x59,0x52,0x4f,0x44,0x75,0x7e,0x63,0x68,
0xb1,0xba,0xa7,0xac,0x9d,0x96,0x8b,0x80,0xe9,0xe2,0xff,0xf4,0xc5,0xce,0xd3,0xd8,
0x7a,0x71,0x6c,0x67,0x56,0x5d,0x40,0x4b,0x22,0x29,0x34,0x3f,0x0e,0x05,0x18,0x13,
0xca,0xc1,0xdc,0xd7,0xe6,0xed,0xf0,0xfb,0x92,0x99,0x84,0x8f,0xbe,0xb5,0xa8,0xa3
};
```

Figure 50: Implementation of Mul11 Lookup Table in C

```
int mul13[256]={
0x00,0x0d,0x1a,0x17,0x34,0x39,0x2e,0x23,0x68,0x65,0x72,0x7f,0x5c,0x51,0x46,0x4b,
0xd0,0xdd,0xca,0xc7,0xe4,0xe9,0xfe,0xf3,0xb8,0xb5,0xa2,0xaf,0x8c,0x81,0x96,0x9b,
0xbb,0xb6,0xa1,0xac,0x8f,0x82,0x95,0x98,0xd3,0xde,0xc9,0xc4,0xe7,0xea,0xfd,0xf0,
0x6b,0x66,0x71,0x7c,0x5f,0x52,0x45,0x48,0x03,0x0e,0x19,0x14,0x37,0x3a,0x2d,0x20,
0x6d,0x60,0x77,0x7a,0x59,0x54,0x43,0x4e,0x05,0x08,0x1f,0x12,0x31,0x3c,0x2b,0x26,
0xbd,0xb0,0xa7,0xaa,0x89,0x84,0x93,0x9e,0xd5,0xd8,0xcf,0xc2,0xe1,0xec,0xfb,0xf6,
0xd6,0xdb,0xcc,0xc1,0xe2,0xef,0xf8,0xf5,0xbe,0xb3,0xa4,0xa9,0x8a,0x87,0x90,0x9d,
0x06,0x0b,0x1c,0x11,0x32,0x3f,0x28,0x25,0x6e,0x63,0x74,0x79,0x5a,0x57,0x40,0x4d,
0xda,0xd7,0xc0,0xcd,0xee,0xe3,0xf4,0xf9,0xb2,0xbf,0xa8,0xa5,0x86,0x8b,0x9c,0x91,
0x0a,0x07,0x10,0x1d,0x3e,0x33,0x24,0x29,0x62,0x6f,0x78,0x75,0x56,0x5b,0x4c,0x41,
0x61,0x6c,0x7b,0x76,0x55,0x58,0x4f,0x42,0x09,0x04,0x13,0x1e,0x3d,0x30,0x27,0x2a,
0xb1,0xbc,0xab,0xa6,0x85,0x88,0x9f,0x92,0xd9,0xd4,0xc3,0xce,0xed,0xe0,0xf7,0xfa,
0xb7,0xba,0xad,0xa0,0x83,0x8e,0x99,0x94,0xdf,0xd2,0xc5,0xc8,0xeb,0xe6,0xf1,0xfc,
0x67,0x6a,0x7d,0x70,0x53,0x5e,0x49,0x44,0x0f,0x02,0x15,0x18,0x3b,0x36,0x21,0x2c,
0x0c,0x01,0x16,0x1b,0x38,0x35,0x22,0x2f,0x64,0x69,0x7e,0x73,0x50,0x5d,0x4a,0x47,
0xdc,0xd1,0xc6,0xcb,0xe8,0xe5,0xf2,0xff,0xb4,0xb9,0xae,0xa3,0x80,0x8d,0x9a,0x97
};
```

Figure 51: Implementation of Mul13 Lookup Table in C

```
int mul14[256]={
0x00,0x0e,0x1c,0x12,0x38,0x36,0x24,0x2a,0x70,0x7e,0x6c,0x62,0x48,0x46,0x54,0x5a,
0xe0,0xee,0xfc,0xf2,0xd8,0xd6,0xc4,0xca,0x90,0x9e,0x8c,0x82,0xa8,0xa6,0xb4,0xba,
0xdb,0xd5,0xc7,0xc9,0xe3,0xed,0xff,0xf1,0xab,0xa5,0xb7,0xb9,0x93,0x9d,0x8f,0x81,
0x3b,0x35,0x27,0x29,0x03,0x0d,0x1f,0x11,0x4b,0x45,0x57,0x59,0x73,0x7d,0x6f,0x61,
0xad,0xa3,0xb1,0xbf,0x95,0x9b,0x89,0x87,0xdd,0xd3,0xc1,0xcf,0xe5,0xeb,0xf9,0xf7,
0x4d,0x43,0x51,0x5f,0x75,0x7b,0x69,0x67,0x3d,0x33,0x21,0x2f,0x05,0x0b,0x19,0x17,
0x76,0x78,0x6a,0x64,0x4e,0x40,0x52,0x5c,0x06,0x08,0x1a,0x14,0x3e,0x30,0x22,0x2c,
0x96,0x98,0x8a,0x84,0xae,0xa0,0xb2,0xbc,0xe6,0xe8,0xfa,0xf4,0xde,0xd0,0xc2,0xcc,
0x41,0x4f,0x5d,0x53,0x79,0x77,0x65,0x6b,0x31,0x3f,0x2d,0x23,0x09,0x07,0x15,0x1b,
0xa1,0xaf,0xbd,0xb3,0x99,0x97,0x85,0x8b,0xd1,0xdf,0xcd,0xc3,0xe9,0xe7,0xf5,0xfb,
0x9a,0x94,0x86,0x88,0xa2,0xac,0xbe,0xb0,0xea,0xe4,0xf6,0xf8,0xd2,0xdc,0xce,0xc0,
0x7a,0x74,0x66,0x68,0x42,0x4c,0x5e,0x50,0x0a,0x04,0x16,0x18,0x32,0x3c,0x2e,0x20,
0xec,0xe2,0xf0,0xfe,0xd4,0xda,0xc8,0xc6,0x9c,0x92,0x80,0x8e,0xa4,0xaa,0xb8,0xb6,
0x0c,0x02,0x10,0x1e,0x34,0x3a,0x28,0x26,0x7c,0x72,0x60,0x6e,0x44,0x4a,0x58,0x56,
0x37,0x39,0x2b,0x25,0x0f,0x01,0x13,0x1d,0x47,0x49,0x5b,0x55,0x7f,0x71,0x63,0x6d,
0xd7,0xd9,0xcb,0xc5,0xef,0xe1,0xf3,0xfd,0xa7,0xa9,0xbb,0xb5,0x9f,0x91,0x83,0x8d
};
```

Figure 52: Implementation of Mul14 Lookup Table in C

C. Vivado Block Design

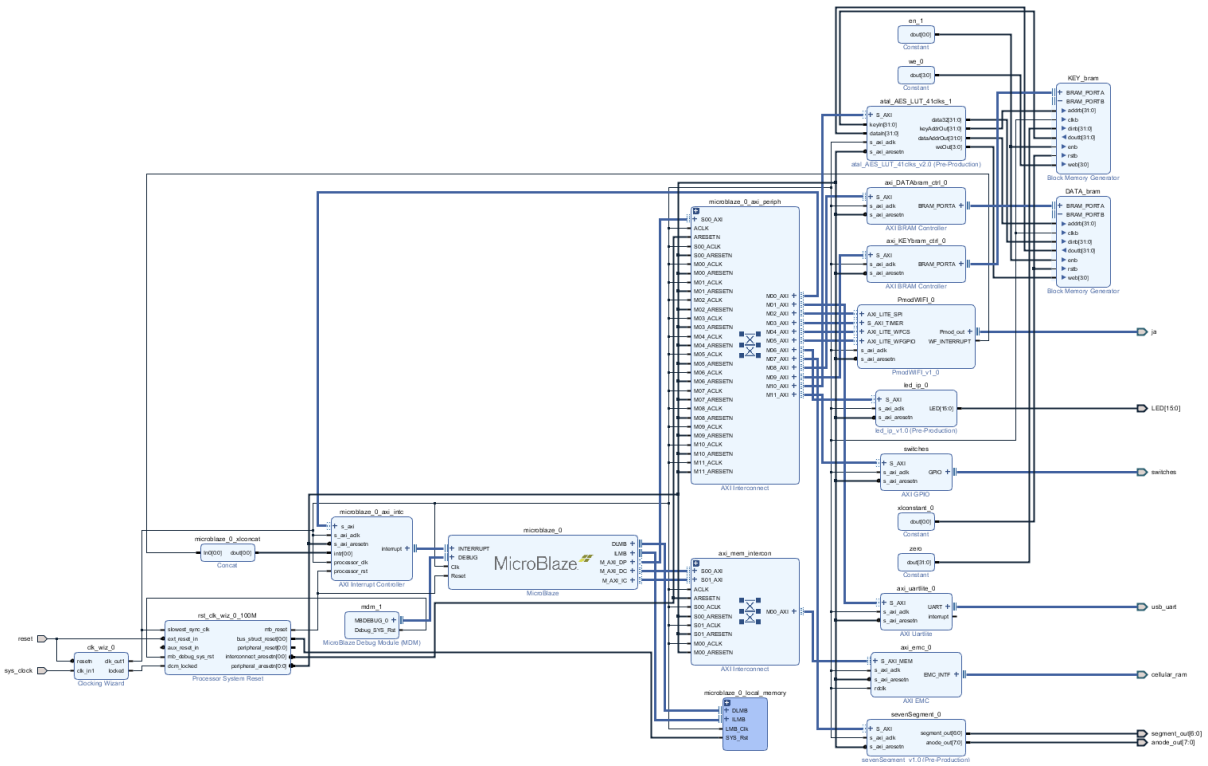


Figure 53: Vivado Block Design of Overall Implementation

D. Block Design Resource Utilization

Resource	Utilization	Available	Utilization %
LUT	27882	63400	43.98
LUTRAM	522	19000	2.75
FF	14641	126800	11.55
BRAM	30	135	22.22
IO	104	210	49.52
BUFG	3	32	9.38
MMCM	1	6	16.67

Figure 54: Resources Utilization Table for Overall Block Design

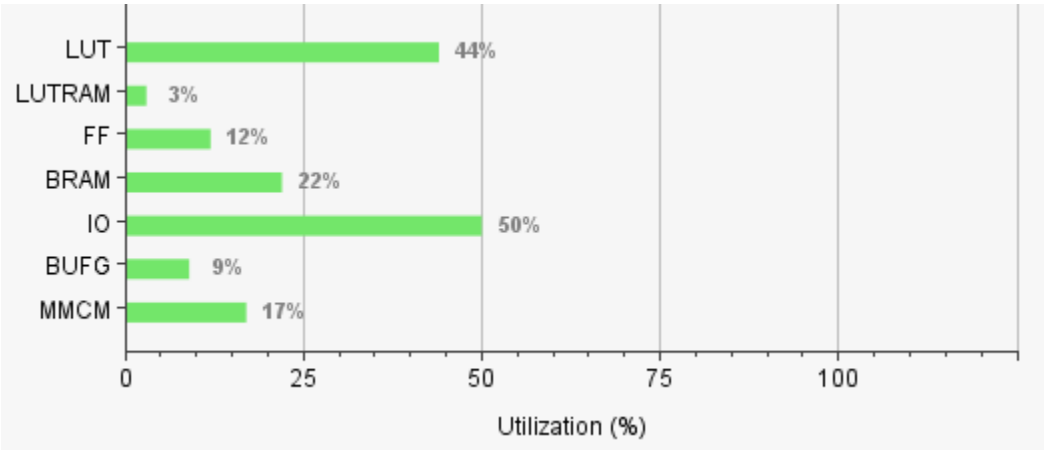


Figure 55: Resources Utilization Graph for Overall Block Design

E. Block Design Power Utilization

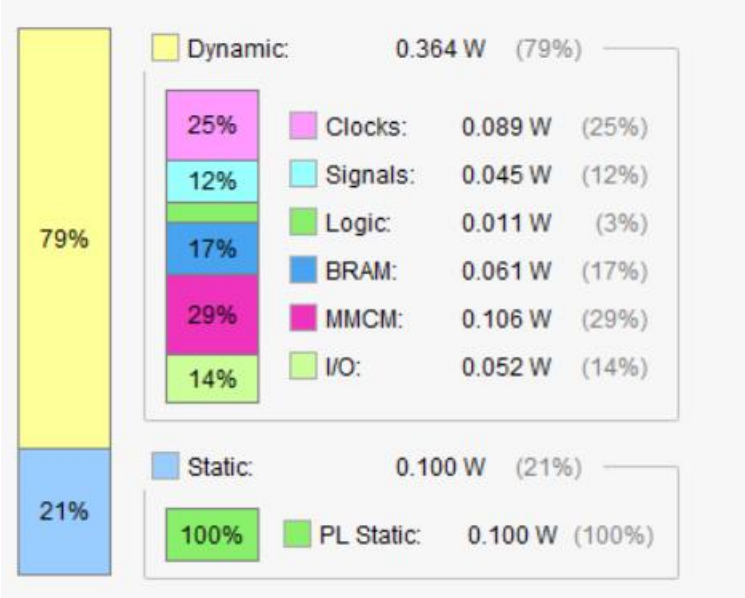


Figure 56: Detailed On-Chip Power Consumption of Overall Block Design

Total On-Chip Power:	**0.464 W**
Junction Temperature:	**27.1 °C**
Thermal Margin:	57.9 °C (12.6 W)
Effective ϑJA:	4.6 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Figure 57: Summary of Power Utilization for Overall Block Design

References

[1] M. Souppaya and K. Scarfone, *Guidelines for Securing Wireless Local Area Networks (WLANs)*, NIST Special Publication 800-153, 2012.

[2] S. Banerji and R. S. Chowdhury, *On IEEE 802.11: Wireless LAN Technology*, International Journal of Mobile Network Communications & Telematics (IJMNCT) , 2013.

[3] J. Edney and W. A. Arbaugh, Real 802.11 Security: Wi-Fi Protected Access and 802.11i, Addison-Wesley, 2004.

[4] Wi-Fi Alliance, *Deploying Wi-Fi Protected Access*, 2005.

[5] Juniper Networks, "Understanding PSK Authentication," Juniper Networks, 16 October 2018. [Online]. Available: https://www.juniper.net/documentation/en_US/junos-space-apps/network-director3.0/topics/concept/wireless-wpa-psk-authentication.html. [Accessed 24 October 2018].

[6] V. Fadyushin, "Common WLAN Protection Mechanisms and their Flaws," 16 March 2016. [Online]. Available: https://hub.packtpub.com/common-wlan-protection-mechanisms-and-their-flaws/. [Accessed 24 October 2018].

[7] T. B. Johnson, "An FPGA architecture for the recovery of WPA/WPA2 keys," Graduate Theses and Dissertations, 2014.

[8] "Understanding WPA/WPA2 Pre-Shared-Key Cracking," 27 December 2015. [Online]. Available: https://www.ins1gn1a.com/understanding-wpa-psk-cracking/. [Accessed 26 October 2018].

[9] O. Nakhila, A. Attiah, Y. Jin and C. Zou, "Parallel Active Dictionary Attack on WPA2-PSK Wi-Fi Networks".

[10] M. Kammerstetter, M. Muellner, D. Burian, C. Kudera and W. Kastner, "Efficient High-Speed WPA2 Brute Force Attacks using Scalable Low-Cost FPGA Clustering," 2016.

[11] A. Visconti, S. Bossi, H. Ragab and A. Calò, *On the weaknesses of PBKDF2*, 2015.

[12] H. Krawczyk, B. M. and R. Canetti, *RFC2104: HMAC: Keyed-Hashing for Message Authentication*, 1997.

[13] H. Yang, "SHA1 Message Digest Algorithm Overview," 2017. [Online]. Available: http://www.herongyang.com/Cryptography/SHA1-Message-Digest-Algorithm-Overview.html. [Accessed 27 October 2018].

[14] National Institute of Standards and Technology(NIST), *Announcing the ADVANCED ENCRYPTION STANDARD (AES),* Federal Information Processing Standards Publication 197, 2001.

[15] A. Berent, *Advanced Encryption Standard by Example.*

[16] J. Monttinen, *The Security of Advanced Encryption Standard*, 2015.

[17] Xilinx, *MicroBlaze Processor Reference Guide*, 2016.

[18] R. Jesman, F. M. Vallina and J. Saniie, *MicroBlaze Tutorial Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software Tools.*

[19] Digilent, *Nexys 4™ FPGA Board Reference Manual,* 2016.

[20] Xilinx, *7 Series FPGAs Data Sheet: Overview*, 2018.

[21] C. Toole, "Introduction to AXI Protocol: Understanding the AXI interface," 2016. [Online]. Available: https://community.arm.com/soc/b/blog/posts/introduction-to-axi-protocol-understanding-the-axi-interface. [Accessed 4 November 2018].

[22] Xilinx, *AXI Reference,* 2011.

[23] Xilimx, *AXI Interconnect v2.1 LogiCORE IP Product Guide,* 2017.

[24] *Vivado Design Suite User Guide: Power Analysis and Optimization,* Xilinx, 2013.

[25] P. V. Kinge, S. Honale and C. Bobade, "Design of AES Algorithm for 128/192/256 Key Length in FPGA," *International Journal of Reconfigurable and Embedded Systems (IJRES),* pp. 49-53, 2014.

[26] V. D. Orlic, M. Peric, Z. Banjac and S. Milicevic, *Some aspects of practical implementation of AES 256 crypto algorithm,* Belgrade: 20th Telecommunications forum TELFOR, 2012.

[27] N. Sai Srinivas and M. Akramuddin, *FPGA Based Hardware Implementation of AES Rijndael Algorithm for Encryption and Decryption,* International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), 2016.

[28] M. Gupta and S. P. Mahto, *Implementation of 128, 192 & 256 bits Advanced Encryption Standard on Reconfigurable Logic,* International Journal of Engineering Trends and Technology (IJETT), 2017.