**Thesis Title Page**

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

FPGA

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

<name of your degree here>

<your academic unit>

<Month of graduation> <Year of graduation>

# **Acknowledgements**

# **Abstract**

The rationale behind the project was to design an efficient implementation of Cryptography algorithms used in Wi-Fi Security protocol for use in wireless modules. The focus of the project was on the efficient software implementation of PBKDF2 (Password-Based Key Derivation Function 2) using HMAC (Keyed-Hash Message Authentication Code)-SHA1,which is used for authentication, and , hardware implementation of AES-256 cipher, which is used for secure data communication, as per IEEE 802.11i Wi-Fi Security (WPA2-PSK) standard. PBKDF2 was implemented using C programming language, while, AES-256 was implemented using Verilog HDL. The overall implementation was designed and tested on Nexys4 FPGA board. Latency (us) was used as the performance metric for PBKDF2, whereas, throughput (Gb/s), resource utilization (Number of Slices), efficiency (GB/s per slice) and latency (ns) were used as performance metrics for AES-256.

The goal of the project was to design an efficient software architecture for PBKDF2 based on HMAC-SHA1, and, hardware architecture in FPGA for the AES-256 cipher, that would adhere to IEEE 802.11i Wi-Fi Security (WPA2-PSK) standard with high throughput and efficiency, and low resource utilization and latency. This hardware when interfaced with a wireless module (MRF24WG0MA PMOD Wi-Fi), was able to communicate with a 2.4 GHz wireless router configured to work in WPA2-PSK security mode. The wireless router was the access point for the communication between multiple wireless end nodes. In this project, the implementation of PBKDF2 using HMAC-SHA1 was used to authenticate our wireless end nodes to a 2.4 GHz wireless router configured in WPA2-PSK security mode, and, implementation of AES-256 was used to create and maintain a secure data communication link.

# **Table of Contents**

[**Acknowledgements** 2](#_Toc527799616)

[**Abstract** 3](#_Toc527799617)

[**Table of Contents** 5](#_Toc527799618)

[**List of Figures** 7](#_Toc527799619)

[**List of Tables** 8](#_Toc527799620)

[**Abbreviations** 9](#_Toc527799621)

[**Chapter 1: Introduction** 10](#_Toc527799622)

[**1.1** **Introduction** 10](#_Toc527799623)

[**1.2** **Project Scope** 14](#_Toc527799624)

[**Chapter 2: Theory** 14](#_Toc527799625)

[**2.1**  **WPA2-PSK Authentication** 14](#_Toc527799626)

[**2.1.1**  **PBKDF2 (Password-Based Key Derivation Function 2)** 15](#_Toc527799627)

[**2.1.2**  **HMAC (Keyed-Hashing for Message Authentication)** 17](#_Toc527799628)

[**2.1.3**  **SHA1** 19](#_Toc527799629)

[**2.2** **WPA2-PSK Data Security** 24](#_Toc527799630)

[**2.2.1** **AES-256 Key Expansion** 24](#_Toc527799631)

[**2.2.2** **AES-256 Encryption** 28](#_Toc527799632)

[**2.2.3**  **AES-256 Decryption** 31](#_Toc527799633)

[**Chapter 3: Implementation** 37](#_Toc527799634)

[**3.1** **WPA2-PSK Implementation** 37](#_Toc527799635)

[**3.1.1**  **Layer1: SHA1-HASH Implementation** 38](#_Toc527799636)

[**3.1.2**  **Layer2: HMAC\_SHA1 Implementation** 40](#_Toc527799637)

[**3.1.3**  **Layer3: PBKDF2 Implementation** 40](#_Toc527799638)

[**3.2** **AES-256 Implementation** 41](#_Toc527799639)

[**3.2.1**  **AES-256 Key Expansion** 42](#_Toc527799640)

[**3.2.2**  **AES-256 Module** 43](#_Toc527799641)

[**Chapter 4: Result and Discussion** 49](#_Toc527799642)

[**4.1** **WPA2-PSK** 49](#_Toc527799643)

[**4.2** **AES-256** 49](#_Toc527799644)

[**References** 51](#_Toc527799645)

# **List of Figures**

[**Figure 1: Example of Wi-Fi Network** 11](#_Toc527656403)

[**Figure 2: WPA2-PSK Security** 12](#_Toc527656404)

[**Figure 3: WPA2-Enterprise Security** 13](#_Toc527656405)

[**Figure 4: Block Diagram for PBKDF2** 16](#_Toc527656406)

[**Figure 5: Block Diagram for HMAC** 18](file:///C:\Thesis\Report\Report_CHP3_1st_Draft.docx#_Toc527656407)

[**Figure 6: Block Diagram for SHA1 Processing Function** 22](file:///C:\Thesis\Report\Report_CHP3_1st_Draft.docx#_Toc527656408)

[**Figure 7: AES-256 Block Diagram** 35](file:///C:\Thesis\Report\Report_CHP3_1st_Draft.docx#_Toc527656409)

[**Figure 8: Layered Software Implementation of WPA2-PSK** 36](file:///C:\Thesis\Report\Report_CHP3_1st_Draft.docx#_Toc527656410)

# **List of Tables**

[**Table 1: S-box Table [6]** 24](#_Toc527656419)

[**Table 2: RCON Table [6]** 24](file:///C:\Thesis\Report\Report_CHP3_1st_Draft.docx#_Toc527656420)

[**Table 3: AES Key Expansion Algorithm [6]** 25](#_Toc527656421)

[**Table 4: State array input and output** 27](#_Toc527656422)

[**Table 5: Table for Mix Column for Encryption** 29](#_Toc527656423)

[**Table 6: Mul2 Table** 29](#_Toc527656424)

[**Table 7: Mul3 Table** 30](#_Toc527656425)

[**Table 8: Inverse S-Box [6]** 31](#_Toc527656426)

[**Table 9: Table for Mix Column for Encryption** 32](#_Toc527656427)

[**Table 10: Mul9 Table** 32](#_Toc527656428)

[**Table 11: Mul11 Table** 33](#_Toc527656429)

[**Table 12: Mul13 Table** 33](#_Toc527656430)

[**Table 13: Mul14 Table** 34](#_Toc527656431)

# **Abbreviations**

# **Chapter 1: Introduction**

## **Introduction**

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 the very mundane to the 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).

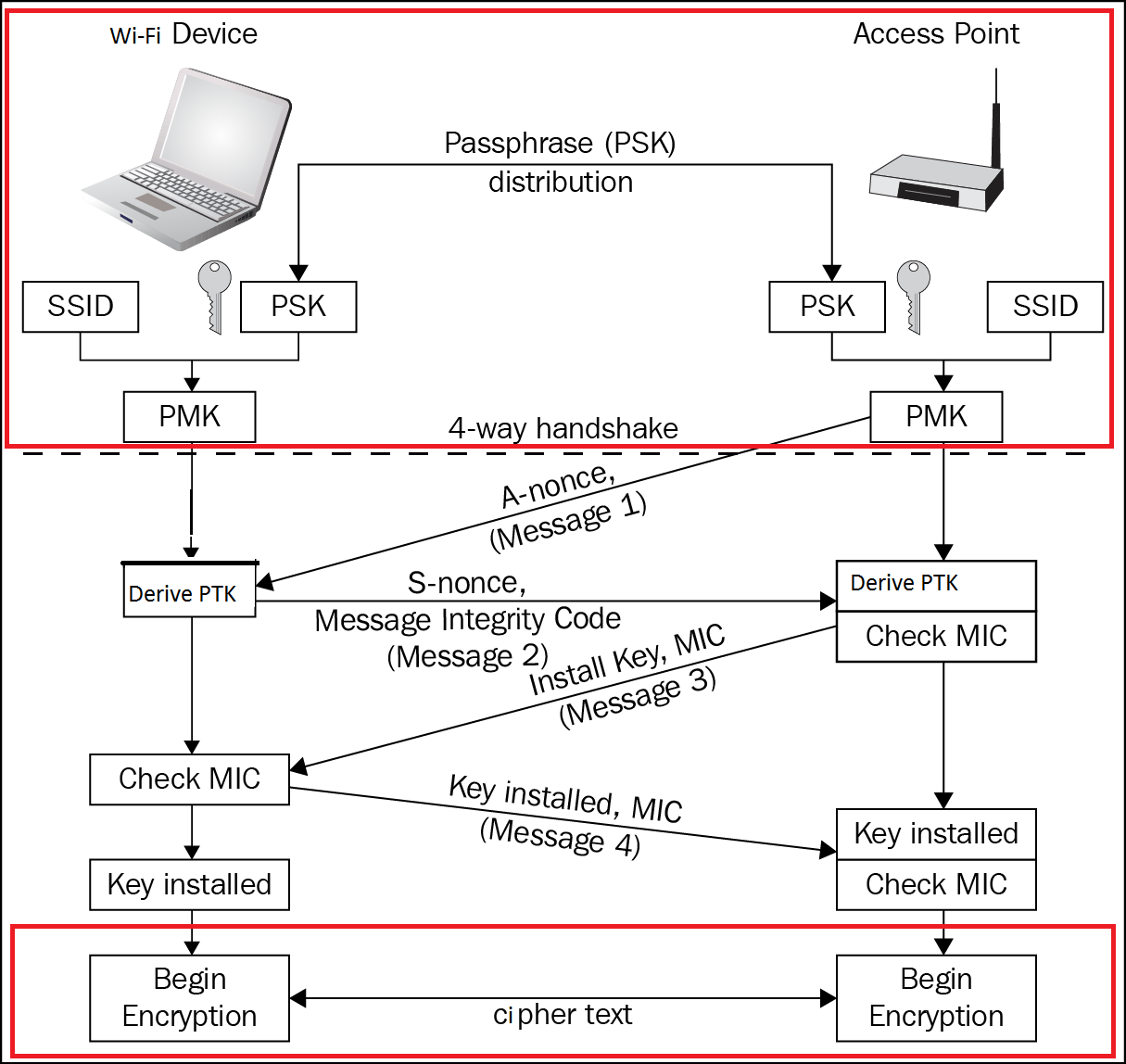
The communication between the devices within a network is governed by a set of rules called communication protocols. The devices within a network must adhere to such protocols to successfully share and interpret data among other devices connected to the network. To maintain interoperability between the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



**Figure 1: Example of Wi-Fi Network**

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.

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 IEEE 802.1X/EAP authentication or PSK technology and includes a new advanced encryption mechanism using the Counter-Mode/CBC-MAC Protocol (CCMP) called the Advanced Encryption Standard (AES) [1]. The IEEE 802.1X/EAP authentication (in enterprise networks) or PSK technology (in personal networks) is used to verify the identity of the communicating wireless devices. Whereas, the AES cipher is used to maintain the confidentiality of the data being communicated. WPA2-Enterprise and WP2-PSK is differentiated by the method it uses for authentication. In WPA2-PSK security, all the end devices have the same pre-shared keys as shown in Figure 2.

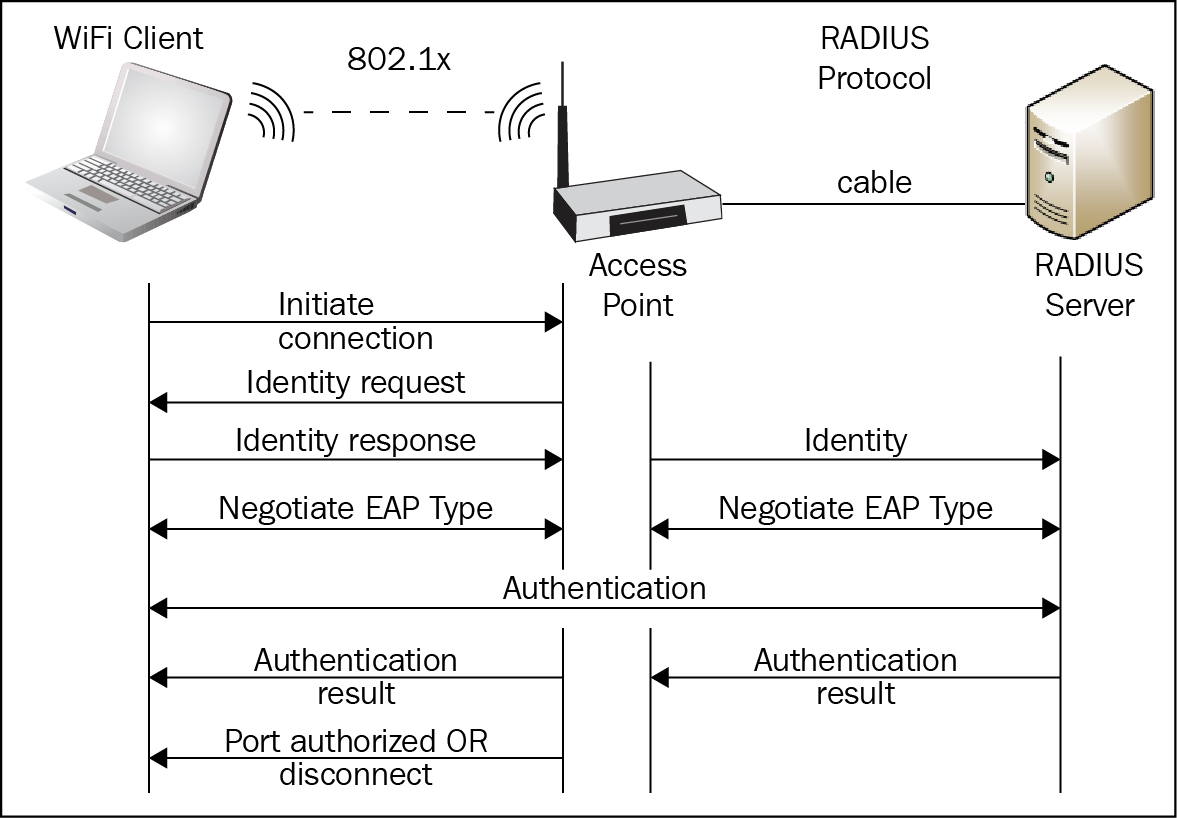


Data Security

Authentication

**Figure 2: WPA2-PSK Security**

While in, WPA2-Enterprise security, an authentication server (RADIUS server) provides individual private keys to the various end nodes as shown in Figure 3.



**Figure 3: WPA2-Enterprise Security**

## **Project Scope**

The primary purpose of this project is to optimize the implementation of security portion of **Wi-Fi** networks. The focus will only be on WPA2-PSK security. To achieve this, the key derivation part of the authentication process, as well as, the AES cipher algorithm required for data security will be optimized. The scope of the optimization will encompass the following areas:

* Efficient software implementation of PBKDF2 based on HMAC-SHA1 for WPA2-PSK authentication.
* Efficient hardware implementation of AES-256 cipher which is used for secure data communication as per the IEEE 802.11i Wi-Fi Security (WPA2-PSK) standard.

# **Chapter 2: Theory**

## **2.1 WPA2-PSK Authentication**

In WPA2-PSK, a single SSID value and a password is assigned to a network. These are hashed to get Pairwise Master key (PMK). This PMK is used during the 4-way handshake process to generate the encryption key for exchanging secure data (Figure 2). In WPA2-PSK, there are only two parties involved in the authentication process, the authenticator (access point) and the supplicant (mobile client). Both parties must prove to each other that they know the pre-shared key to ensure a secure connection. The pre-shared key is never exchanged between the supplicant and authenticator as the channel of communication is not secure before the authentication process has completed. Hence, without this level of authentication, sharing of the pre-shared key would be done through an unencrypted channel and susceptible to be discovered by outside parties. We thus use PBKDF2 (Password-Based Key Derivation Function 2) to overcome this.

### **2.1.1 PBKDF2 (Password-Based Key Derivation Function 2)**

PBKDF2 uses a pseudorandom function to derive keys. Though length of the derived key is essentially unbounded, however, the maximum effective search space for the derived key is limited by the structure of the underlying pseudorandom function [3]. The PBKDF2 key derivation function is defined as follow:

DK = PBKDF2(PRF, P, S, C, dkLen) ................... (1)

where,

DK: derived key

PRF:  pseudorandom function of two parameters with output length hLen

P: password

S: salt (sequence of bits)

C: iteration count, a positive integer

dkLen: intended length in octets of the derived key, a positive integer, at most (232 - 1) \* hLen

To derive key from PBKDF2, each hLen bit block Ti of derived key DK, is computed as follows:

DK = T1 || T2 || ... || Tdklen/hlen ..................... (2)

Ti = F (Password, Salt, C, i) ...................... (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 4.

F (Password, Salt, C, i) = U1 ⊕ U2 ⊕ ... ⊕ Uc .... (4)

where,

U1 = PRF (Password, Salt || INT\_32\_BE(i)) ......... (5)

U2 = PRF (Password, U1) ........................... (6)

...

Uc = PRF (Password, Uc - 1) ......................... (7)



**Figure 4: Block Diagram for PBKDF2**

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

DK = PBKDF2(HMAC−SHA1, pwd, ssid, 4096, 32) ...... (8)

### **2.1.2 HMAC (Keyed-Hashing for Message Authentication)**

Hash Based Message Authentication Code (HMAC) provides a mechanism to calculate a message authentication code (MAC) based around a cryptographic hashing function [4]. 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 [5]. The definition of HMAC requires a cryptographic hash function denoted by H, with block size B bytes and output length L bytes, and a secret key K [5]. The authentication key K can be of any length up to B. Applications that use keys longer than B bytes will first hash the key using H and then use the resultant L byte string as the actual key to HMAC [5]. In case of WPA2, values of B and L are 64 bytes and 20 bytes respectively. The HMAC function is defined as follows:

HMAC (K, m) = H ((K' ⊕ opad) || H ((K' ⊕ ipad)

|| m)) ..........(9)

where,

H: a cryptographic hash function

K: the secret key

m: the message to be authenticated

K': another secret key, derived from the original key K

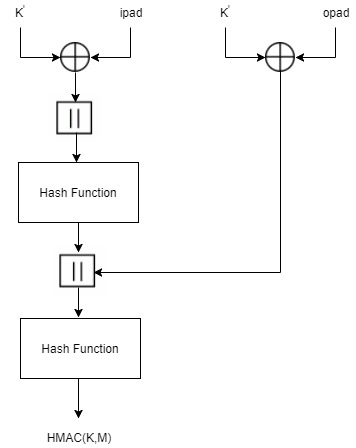
(by padding K to the right with extra zeroes to the

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 5.



**Figure 5: Block Diagram for HMAC**

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

HMAC-SHA1 (pwd, ssid) = SHA1 ((pwd ⊕ opad) || SHA1((pwd

⊕ opad ⊕ ipad) ||ssid) ... (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 [4]. SHA1 algorithm primarily consists of 6 steps [2]:

**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 of SHA1. These constants are as follows:

for 0 <= r <= 19,

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

for 20 <= t <= 39,

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

for 40 <= t <= 59,

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

for, 60 <= t <= 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 6.



**Figure 6: Block Diagram for SHA1 Processing Function**

## **2.2 WPA2-PSK Data Security**

WPA2-PSK uses Advanced Encryption Standard (AES) cipher for data security. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext and decrypting the ciphertext converts the data back into its original form, called plaintext [7]. AES has block size of 128 bits and a key size of 128, 192 or 256 bits. WPA2-PSK uses key size of 256 bits i.e. AES-256. AES-256 requires 60 rounds for expansion and the size of the expanded key is 240 bytes. It requires 14 rounds for the completion of both encryption and decryption process. The size of the encrypted data is 128 bits.

### **2.2.1 AES-256 Key Expansion**

Before the encryption and decryption process the 256 bits key of AES-256 is expanded to get a 240-byte expanded key value. The key expansion routine executes a maximum of four consecutive functions [6]. They are as follows:

* **Rot Word ():** This function takes 4 bytes as an argument. It performs circular left shift on 4 bytes.

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

* **Sub Word ():** This function takes 4 bytes as an argument. It performs S-box substitution operation on those 4 bytes. The S-box substitution table is shown in Table 1.
* **Rcon ():** This function returns a 4-byte value based on the Table 2.
* **EK (Offset):** This function takes offset as the argument and returns 4 bytes of the expanded key after offset. For example, if offset is 0, then, EK () will return bytes 0, 1, 2,3 of the expanded key.
* **K (Offset):** This function takes offset as the argument and returns 4 bytes of the key after offset. For example, if offset is 0, then, K () will return bytes 0,1,2,3 of the key.

**Table 1: S-box Table [6]**





**Table 2: RCON Table [6]**

The expanded key value can be obtained by following Table 3 [6].

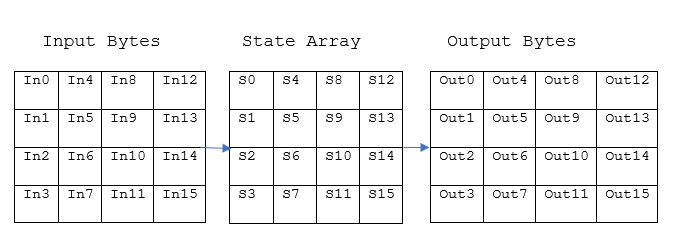
**Table 3: AES Key Expansion Algorithm [6]**

|  |  |  |
| --- | --- | --- |
| **Round** | **Expanded Key Bytes** | **Function** |
| 0 | 0 1 2 3 | K (0) |
| 1 | 4 5 6 7 | K (4) |
| 2 | 8 9 10 11 | K (8) |
| 3 | 12 13 14 15 | K (12) |
| 4 | 16 17 18 19 | K (16) |
| 5 | 20 21 22 23 | K (20) |
| 6 | 24 25 26 27 | K (24) |
| 7 | 28 29 30 31 | K (28) |
| 8 | 32 33 34 35 | Sub Word(Rot Word(EK((8-1)\*4))) XOR Rcon((8/8)-1) XOR EK((8-8)\*4) |
| 9 | 36 37 38 39 | EK((9-1)\*4)XOR EK((9-8)\*4) |
| 10 | 40 41 42 43 | EK((10-1)\*4)XOR EK((10-8)\*4) |
| 11 | 44 45 46 47 | EK((11-1)\*4)XOR EK((11-8)\*4) |
| 12 | 48 49 50 51 | Sub Word(EK((12-1)\*4))XOR EK((12-8)\*4) |
| 13 | 52 53 54 55 | EK((13-1)\*4)XOR EK((13-8)\*4) |
| 14 | 56 57 58 59 | EK((14-1)\*4)XOR EK((14-8)\*4) |
| 15 | 60 61 62 63 | EK((15-1)\*4)XOR EK((15-8)\*4) |
| 16 | 64 65 66 67 | Sub Word(Rot Word(EK((16-1)\*4))) XOR Rcon((16/8)-1) XOR EK((16-8)\*4) |
| 17 | 68 69 70 71 | EK((17-1)\*4)XOR EK((17-8)\*4) |
| 18 | 72 73 74 75 | EK((18-1)\*4)XOR EK((18-8)\*4) |
| 19 | 76 77 78 79 | EK((19-1)\*4)XOR EK((19-8)\*4) |
| 20 | 80 81 82 83 | Sub Word(EK((20-1)\*4))XOR EK((20-8)\*4) |
| 21 | 84 85 86 87 | EK((21-1)\*4)XOR EK((21-8)\*4) |
| 22 | 88 89 90 91 | EK((22-1)\*4)XOR EK((22-8)\*4) |
| 23 | 92 93 94 95 | EK((23-1)\*4)XOR EK((23-8)\*4) |
| 24 | 96 97 98 99 | Sub Word(Rot Word(EK((24-1)\*4))) XOR Rcon((24/8)-1) XOR EK((24-8)\*4) |
| 25 | 100 101 102 103 | EK((25-1)\*4)XOR EK((25-8)\*4) |
| 26 | 104 105 106 107 | EK((26-1)\*4)XOR EK((26-8)\*4) |
| 27 | 108 109 110 111 | EK((27-1)\*4)XOR EK((27-8)\*4) |
| 28 | 112 113 114 115 | Sub Word(EK((28-1)\*4))XOR EK((28-8)\*4) |
| 29 | 116 117 118 119 | EK((29-1)\*4)XOR EK((29-8)\*4) |
| 30 | 120 121 122 123 | EK((30-1)\*4)XOR EK((30-8)\*4) |
| 31 | 124 125 126 127 | EK((31-1)\*4)XOR EK((31-8)\*4) |
| 32 | 128 129 130 131 | Sub Word(Rot Word(EK((32-1)\*4))) XOR Rcon((32/8)-1) XOR EK((32-8)\*4) |
| 33 | 132 133 134 135 | EK((33-1)\*4)XOR EK((33-8)\*4) |
| 34 | 136 137 138 139 | EK((34-1)\*4)XOR EK((34-8)\*4) |
| 35 | 140 141 142 143 | EK((35-1)\*4)XOR EK((35-8)\*4) |
| 36 | 144 145 146 147 | Sub Word(EK((36-1)\*4))XOR EK((36-8)\*4) |
| 37 | 148 149 150 151 | EK((37-1)\*4)XOR EK((37-8)\*4) |
| 38 | 152 153 154 155 | EK((38-1)\*4)XOR EK((38-8)\*4) |
| 39 | 156 157 158 159 | EK((39-1)\*4)XOR EK((39-8)\*4) |
| 40 | 160 161 162 163 | Sub Word(Rot Word(EK((40-1)\*4))) XOR Rcon((40/8)-1) XOR EK((40-8)\*4) |
| 41 | 164 165 166 167 | EK((41-1)\*4)XOR EK((41-8)\*4) |
| 42 | 168 169 170 171 | EK((42-1)\*4)XOR EK((42-8)\*4) |
| 43 | 172 173 174 175 | EK((43-1)\*4)XOR EK((43-8)\*4) |
| 44 | 176 177 178 179 | Sub Word(EK((44-1)\*4))XOR EK((44-8)\*4) |
| 45 | 180 181 182 183 | EK((45-1)\*4)XOR EK((45-8)\*4) |
| 46 | 184 185 186 187 | EK((46-1)\*4)XOR EK((46-8)\*4) |
| 47 | 188 189 190 191 | EK((47-1)\*4)XOR EK((47-8)\*4) |
| 48 | 192 193 194 195 | Sub Word(Rot Word(EK((48-1)\*4))) XOR Rcon((48/8)-1) XOR EK((48-8)\*4) |
| 49 | 196 197 198 199 | EK((49-1)\*4)XOR EK((49-8)\*4) |
| 50 | 200 201 202 203 | EK((50-1)\*4)XOR EK((50-8)\*4) |
| 51 | 204 205 206 207 | EK((51-1)\*4)XOR EK((51-8)\*4) |
| 52 | 208 209 210 211 | Sub Word(EK((52-1)\*4))XOR EK((52-8)\*4) |
| 53 | 212 213 214 215 | EK((53-1)\*4)XOR EK((53-8)\*4) |
| 54 | 216 217 218 219 | EK((54-1)\*4)XOR EK((54-8)\*4) |
| 55 | 220 221 222 223 | EK((55-1)\*4)XOR EK((55-8)\*4) |
| 56 | 224 225 226 227 | Sub Word(Rot Word(EK((56-1)\*4))) XOR Rcon((56/8)-1) XOR EK((56-8)\*4) |
| 57 | 228 229 230 231 | EK((57-1)\*4)XOR EK((57-8)\*4) |
| 58 | 232 233 234 235 | EK((58-1)\*4)XOR EK((58-8)\*4) |
| 59 | 236 237 238 239 | EK((59-1)\*4)XOR EK((59-8)\*4) |

### **2.2.2 AES-256 Encryption**

Internally, the AES algorithm’s operations are performed on a two-dimensional array of bytes called the **State** [7]. For AES-256, the State consists of four rows of bytes, each containing 4 bytes. At the start of the Cipher and Inverse Cipher, the input – the array of bytes In0, In1, … In15 – is copied into the State array as illustrated in Table 4. After an initial Round Key addition, the State array is transformed by implementing a round function 14 times , with the final round differing slightly from the first 13 rounds. The Cipher or Inverse Cipher 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 [7].

**Table 4: State array input and output**



AES is an iterated block cipher, which means, the same operations are performed many times on a fixed number of bytes. Hence, these operations can easily be broken down to the following functions [6]:

* **Add Round Key ():** Exclusive-OR operation is performed between each 16 bytes of a state with the 16 bytes portion of the expanded key for the current round. The expanded key bytes are only used once for a block of plaintext. In the next round, Exclusive-OR operation is performed between the next 16 bytes of the expanded keys and 16 bytes of the next state.
* **Byte Sub ():** In this stage, each byte of data is substituted with the corresponding value from the S-box table (Table 1).
* **Shift Row ():** 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.
* **Mix Column ():** The matrix obtained from the Shift Row () operation goes through the multiplication over Galois Field (Table 5) . The multiplication over Galois Field lookup tables involved in theMix Column () operation are shown in Table 6 and Table 7.

**Table 5: Table for Mix Column for Encryption**



**Table 6: Mul2 Table**



**Table 7: Mul3 Table**



### **2.2.3 AES-256 Decryption**

AES decryption process goes through similar stages as AES encryption. The decryption process uses the following 4 functions:

* **Add Round Key ():** The order of the expanded key is reversed for decryption. Exclusive-OR operation is first performed with the last 16 bytes of the expanded key, then the second last 16 bytes and so on.
* **Inverse Byte Sub ():** In this stage, each byte of data is substituted with the corresponding value from the inverse S-box table (Table 7).

**Table 8: Inverse S-Box [6]**



* **Inverse Shift Row ():** 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.
* **Inverse Mix Column ():** The matrix obtained from the Inverse Shift Row () operation goes through the multiplication over Galois Field (Table 9) . The multiplication over Galois Field lookup tables involved in theInverse Mix Column () operation are shown in Table 10, Table 11, Table 12, Table 13.

**Table 9: Table for Mix Column for Encryption**



**Table 10: Mul9 Table**



**Table 11: Mul11 Table**



**Table 12: Mul13 Table**



**Table 13: Mul14 Table**



The complete block diagram for AES-256 is shown in Figure 7.



**Figure 7: AES-256 Block Diagram**

# **Chapter 3: Implementation**

## **3.1 WPA2-PSK Implementation**

The code for WPA2-PSK was written using C programming language and for MicroBlaze soft Microprocessor core running on the Nexys4 FPGA board. The overall code for WPA2-PSK implementation contains 4 layers of code. At the bottom layer, there is the 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 11. This code was written for 32-bit soft processor Micro Blaze.



**Figure 8: Layered Software Implementation of WPA2-PSK**

### **3.1.1 Layer1: SHA1-HASH Implementation**

This layer deals 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

This data type is used to maintain information relevant to a particular iteration of an SHA1-Hash process. It holds 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. The actual C code definition for this data type is given below:

typedef struct {

uint8\_t data [64];

uint32\_t datalen;

unsigned long long bitlen;

uint32\_t state [5];

} SHA1\_CTX;

* Function Prototype 1: void SHA1Init (SHA1\_CTX \*context)

This Function is used to initialize a new context for the SHA1-Hash process. It initializes datalen and bitlen to 0. It also initializes the states of the context to the initial Hash values of 0x67452301 ,0xEFCDAB89, 0x98BADCFE,

0x10325476 and 0xC3D2E1F0.

* Function Prototype 2: void SHA1Update (SHA1\_CTX \*context,

const void \*data, uint32\_t len)

This function is used to update data, data length and bit length for the context of SHA-1. If the data length is 64 bytes (512 bits) i.e. input block size of SHA-1, we start the SHA1-Hash process by calling SHA-1 transform ().

* Function Prototype 3: void SHA1Final (unsigned char digest [20],

SHA1\_CTX\* context);

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

* Function Prototype 4: SHA1Transform (SHA1\_CTX \*ctx,

const uint8\_t data [])

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

### **3.1.2 Layer2: HMAC\_SHA1 Implementation**

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

Function Prototype: void my\_hmac\_sha1(const unsigned char \*text,

int text\_len, const unsigned char \*key,

int key\_len, unsigned char \*digest)

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

### **3.1.3 Layer3: PBKDF2 Implementation**

This layer deals with the function related to the implementation of PBKDF2 operation. It contains a single function that calls my\_hmac\_sha1() for 4096 iterations after doing some initial processing. The prototype for this function is given below:

Function Prototype: int pkcs5\_pbkdf2(const char \*pass, const uint8\_t \*salt,

size\_t salt\_len, unsigned int rounds, uint8\_t \*key, size\_t key\_len)

We will get the 256-bit master key used as the first key in AES encryption from this function. The application code will use this function to obtain the Hashed key from SSID-Passphrase combination, which is used to authenticate a wireless device using WPA2-PSK before starting the wireless communication.

## **3.2 AES-256 Implementation**

The implementation of AES-256 consists of 2 main modules: Key Expansion Module and AES Module. The key expansion module was used to derive the expanded key set from the 256-bit (32 byte) main key which was obtained from the PBKDF2 authentication process. The AES core contains modules for encryption and decryption blocks. These blocks used the expanded key to convert plaintext to ciphertext and vice versa.

The code for key expansion portion of AES-256 implementation was written using C programming language. In the key expansion process, a set of 240 bytes of expanded key were derived from the main key . The 240 bytes of keys were divided into sets of 16 bytes, which were used at each round of AES transformation (Encryption and Decryption). In this implementation, two different sets of 240 bytes of expanded keys were derived for encryption and decryption. This was done because the decryption block of the AES-256 was implemented using “Equivalent Inverse Cipher Method”. This method of decryption was used in the design because, in this method, both the encryption and decryption processes have the same sequence of transformations with the decryption process having normal transformations replaced by their inverses. In doing so, the performance of both encryption and decryption blocks would be the same. These expanded keys were stored in BRAM of the FPGA. 4 Bytes of keys were stored per BRAM location of FPGA. Hence, 60 memory locations were occupied to store 240 bytes of expanded key for encryption. Similarly, further 60 memory locations were occupied 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 from the temporary buffer instead of BRAM to improve the performance.

The encryption and decryption portion of AES-256 implementation was written using Verilog HDL. The block size of AES is 128 bits (16 Bytes). Hence, both encryption and decryption process work on 128-bits (16 Bytes) of data at a time. If the data to be transformed is less than 128 bits, then, it is padded with trailing 0’s to make it 128-bit block before being transformed. If the data to be transformed is greater than 128 bits but not a multiple of 128 bits, then, it is also padded with trailing 0’s until we have 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 are 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.

### **3.2.1 AES-256 Key Expansion**

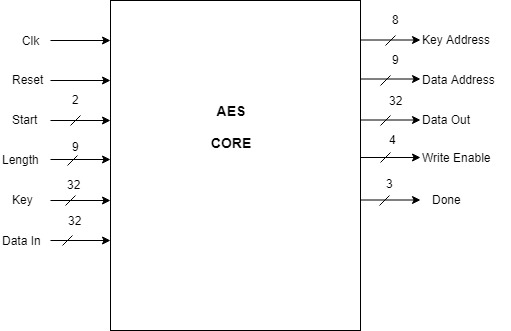
The 256-bit (32 byte) key obtained from the PBKDF2 during the Authentication process is used as the main key for AES-256 expansion. The main key goes through the key expansion process at the end of which further 13 set of 16-byte keys are obtained.

### **3.2.2 AES-256 Module**

The implementation of AES-256 Module consisted of 2 components: AES core and BRAMs (Key BRAM and Data BRAM).

#### **3.2.2.1 AES Core**

The block diagram of AES-256 core implementation is shown in Figure. It has the following port definition:



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

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

* + Length: It is a 9-bit input value which signifies 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 is a 32-bit input value which signifies the expanded key input to the AES core. This input port was connected to 32-bit Data output port of Key BRAM.
  + Data In: It is a 32-bit input value which signifies the 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 is a 32-bit output value which denotes the memory location of the Key BRAM from which the expanded key is to be retrieved during the encryption/decryption process. On reset the value of Key Address is 0. This output port was connected to 32-bit address input port of Key BRAM.
  + Data Address: It is a 32-bit output value which denotes the memory location of the Data BRAM from which the expanded key is to be retrieved during the encryption/decryption process. On reset the value of Data Address is 0.

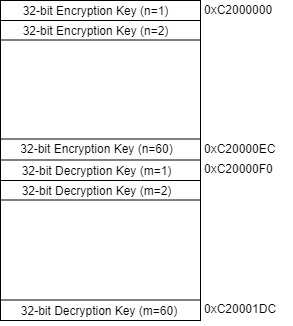
This output port was connected to 32-bit address input port of Data BRAM.

* + Data Out: It is a 32-bit output value which signifies ciphertext in case of encryption operation and plaintext in case of decryption operation. On reset the value of Data Out is 0. This output port was connected to 32-bit Data input port of Data BRAM.
  + Write Enable: It is a 4-bit output signal that is 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 is 0. This output port was connected to 4-bit Write Enable input port of Data BRAM.
  + Done: It is a 3-bit output signal that shows the completion of the AES operation. The value of Done signal becomes 3’b111 after the AES transformation is completed and the new data is written into the allocated memory locations in the Data BRAM. On reset the value of Done is 0. This output signal was connected to bits[2:0] of Slave register1 from the AXI BUS of MicroBlaze.

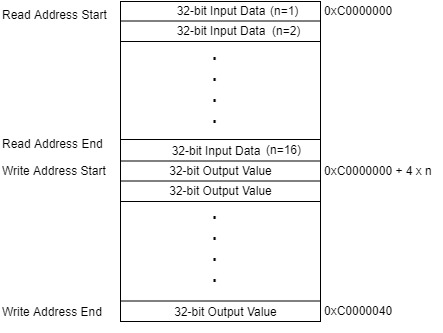
#### **3.2.2.2 BRAMs**

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.

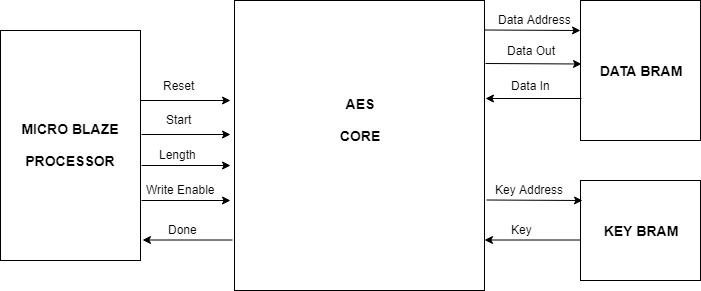
* Key BRAM: It is a Read Only Memory block where expanded keys for encryption and decryption are stored. The Data width of the Key BRAM is 32-bits. 60 such memory locations were allocated each for encryption decryption keys. Hence, 480 bytes of BRAM memory was used for storing the expanded keys. The start address for Key BRAM was 0xC2000000. The overall memory organization for Key BRAM is shown in Figure.



* Data BRAM: It is a Read/Write Memory block where original data and data to be transformed are stored. The Data width of the Data BRAM is 32-bits. 16 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.



The overall block diagram for interfacing AES core with BRAMs and MicroBlaze processor is shown in Figure.

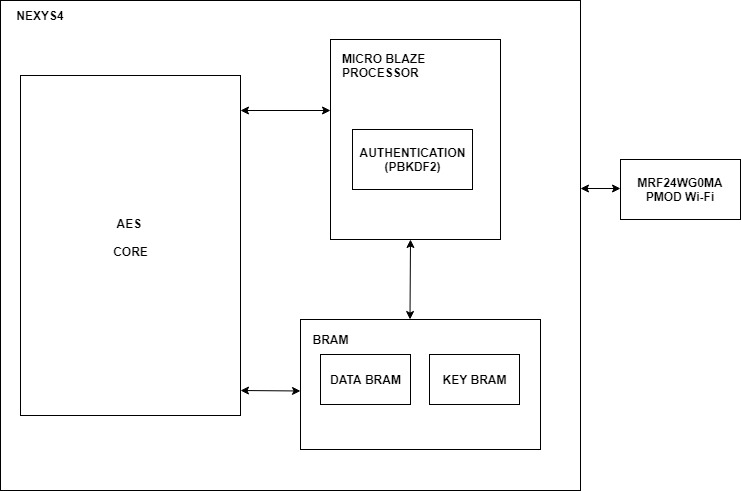


# **Chapter 4: Result and Discussion**

## **4.1 WPA2-PSK**

The test for authentication of Wi-Fi using WPA2-PSK was performed with both wireless router and mobile hotspot as the access point. Two wireless modules (MRF24WG0MA PMOD Wi-Fi) were used as end nodes to connect to the access point on both cases of the access point. The SSID and Password of the access point were known beforehand. When SSID and Password information were correctly entered in the end nodes, the access point was able to successfully authenticate them. After successful authentication , the access point was able to create a communication channel between itself and the end nodes. For communication between multiple end nodes, the data goes through the access point. When incorrect values of SSID and/or Password was entered on the end nodes, authentication failed and a communication channel was not created.

## **4.2 AES-256**



# **References**

1. WPA and WPA2 Implementation White Paper, “Deploying Wi-Fi Protected Access

(WPA™) and WPA2™ in the Enterprise”, March 2005

1. Yang, Herong, “Cryptography Tutorials - Herong's Tutorial Examples”, Retrieved

September 11, 2017

[http://www.herongyang.com/Cryptography/SHA1-Message-Digest-Algorithm-](http://www.herongyang.com/Cryptography/SHA1-Message-Digest-Algorithm-   Overview.html)

[Overview.html](http://www.herongyang.com/Cryptography/SHA1-Message-Digest-Algorithm-   Overview.html)

1. Kaliski, B. “RFC 2898: PKCS #5: Password-Based Cryptography Specification Version 2.0”, September 2000
2. Johnson, Tyler Blaine, “An FPGA architecture for the recovery of WPA/WPA2 keys”, 2014
3. Krawczyk, H., Bellare M., Canetti, “RFC2104: HMAC: Keyed-Hashing for Message Authentication”, February 1997
4. Berent, Adam, “Advanced Encryption Standard by Example”
5. Federal Information Processing Standards Publication 197, “Advanced Encryption Standard (AES)”, November 26, 2001

Debasis Das

1

, Rajiv Misra

2