Block cipher four implementation on field programmable gate array

Yusuf Kurniawan\textsuperscript{a,*,} Muhammad Adli Rizqulloh\textsuperscript{b}

\textsuperscript{a}Electrical Engineering, School of Electrical Engineering and Informatics, Institut Teknologi Bandung, Bandung City, 40132, Indonesia
\textsuperscript{b}Department of Electrical Engineering Education, Universitas Pendidikan Indonesia, Bandung City, 40154, Indonesia

Article history:
Received: 9 June 2020 / Received in revised form: 12 August 2020 / Accepted: 23 August 2020

Abstract

Block ciphers are used to protect data in information systems from being leaked to unauthorized people. One of many block cipher algorithms developed by Indonesian researchers is the BCF (Block Cipher-Four) - a block cipher with 128-bit input/output that can accept 128-bit, 192-bit, or 256-bit keys. The BCF algorithm can be used in embedded systems that require fast BCF implementation. In this study, the design and implementation of the BCF engine were carried out on the FPGA DE2. It is the first research on BCF implementation in FPGA. The operations of the BCF machine were controlled by Nios II as the host processor. Our experiments showed that the BCF engine could compute 2,847 times faster than a BFC implementation using only Nios II / and e. Our contribution presents the description of new block cipher BCF and the first implementation of it on FPGA using an efficient method.

Keywords: BCF; FPGA; NIOS II; Cryptography

1. Introduction

Block cipher is one of the cryptographic components used to protect information. Information can be in the internet network, financial system, military, and IoT (internet of things). IoT is a network of interconnected objects in various forms such as wireless sensor networks, electrical, electronic, mechanical devices, and their interaction with computer data via the internet [5]. In the IoT period, embedded devices were connected to the internet. The advent of IoT has put telecommunications and embedded systems at risk [6]. BCF is an encryption algorithm based on AES [13], Camellia [14], TwoFish [15], and Khazad [12]. It has 128 bits of input/output and 128, 192, and 256 bits keys. BCF is an encryption algorithm designed by Indonesian researchers [1]. This algorithm has an advantage over AES: The key schedule in BCF is more secure than AES because the main key is very difficult to find even when all sub-keys of BCF have been found. The SBox from BCF changes dependent on the key, while the SBox from AES does not change. Thus, BCF is safer than AES.

There are two types of BCF keys: master key and sub-keys. A master key is processed by key schedule function becoming the sub-keys. Every sub-key is used to encrypt or decrypt partial data in every round. Encryption is a process to convert plaintext to be cipher text and decryption converts cipher text to be plaintext.

Cryptanalysis is used to crack the key of a block cipher in an unusual way or test the security of a cryptographic algorithm that has been created. Correlation power analysis, for instance, tries to find all of the sub-keys using the correlation between the hamming weights and the power used in the embedded device when calculating the encryption algorithm [7].

The hardware implementation is very important in terms of a performance and security, especially as a countermeasure against timing attacks [8] in particular and as side-channel attacks in general. This paper aims to introduce the BCF algorithm implemented in FPGA with an efficient method. This paper proposes a hardware architecture of the BCF algorithm as a co-host processor (encryption engine accelerator). This architecture was written in Verilog and tested on the Altera Cyclone IV EP4CE115F29 [9] using NIOS as the host processor. We compared the results with AES, Camellia, and TDEA data taken from SASEBO [10]. Moreover, we compared the BCF hardware accelerator with software implementation enabling us to measure how fast the BCF encryption engine accelerator computed, compared to software.

2. Materials and Methods

2.1. BCF Algorithm

BCF uses the Feistel structure [11], in contrast to AES which uses the SPN structure. The SPN structure requires fewer rounds than does Feistel to achieve the same diffusion rate. The advantage of using the Feistel structure over SPN is related to the use of the same structure for the encryption and decryption.
processes so that it will require few memories in the implementation. SPN requires two different algorithms for encryption and decryption.

The BCF algorithm has two main components: scheduling part and randomization part. Key Schedule is performed to generate sub keys and randomization is performed to encrypt or decrypt data using sub keys generated by key scheduling. The number of rounds at the randomization stage depends on the length of the key in which 128-bit keys are used in the randomization of 15 rounds, 192-bit keys require 16 rounds and 256-bit keys for 18 rounds. In each round, the F0 function is applied. This function uses sub keys to manipulate the input data for each round.

The main features of the BCF algorithm are:

1. The input and output data are 128 bits (plain text and cipher text) respectively.
2. The length of the master key has 3 variants: 128, 192 and 256 bits.
3. Key scheduling is done in 8 rounds using the F0 function.
4. The number of rounds at the randomizing stage (for encryption or decryption) depends on the length of the key.

The key schedule stage is carried out at the beginning to generate sub-keys for the randomizing stage, but in this paper we will begin by explaining the randomizing stage.

2.2. BCF Encryption

BCF uses the Feistel structure as in the Twofish algorithm, so it can use the same algorithm for encryption and decryption. BCF has 128-bit input / output. The pseudo code of the BCF encryption algorithm is presented as follows.

\[
\begin{align*}
&\text{XL} = \text{XL} \ ^\land \ \text{KW1} \\
&\text{XR} = \text{XR} \ ^\land \ \text{KW2} \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K1},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K2},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K3},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K4},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K5},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K6},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K7},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K8},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K9},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K10},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K11},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K12},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K13},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K14},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K15},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K16},\text{XR}) \\
&\text{XL} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K17},\text{XR}) \\
&\text{XR} = \text{XR} \\
&\text{XR} = \text{XL} \ ^\land \ F0(\text{K18},\text{XR}) \\
&\text{XL} = \text{XL} \ ^\land \ \text{KW3} \\
&\text{XR} = \text{XR} \ ^\land \ \text{KW4}
\end{align*}
\]

Fig. 1 shows the BCF structure that can be used for both encryption and decryption.

In the BCF algorithm, there is an FO function that has an input of two data words x and two words of the k sub-key and produces two output words of y, where 1 word is 32 bits. This function is the heart of BCF encryption/decryption. For a note, 1 word is 32 bits.

\[ y = FO(x,k) = P(S(x)) \oplus k \]  

SBox has 1 byte input / output. Because each x consists of 8 bytes, there are 8 SBox operations for each input \(x\) and k sub-key and produces two output words of y, where 1 word is 32 bits. This function is the heart of BCF encryption/decryption. For a note, 1 word is 32 bits.

\[ y = FO(x,k) = P(S(x)) \oplus k \]  

SBox has 1 byte input / output. Because each x consists of 8 bytes, there are 8 SBox operations for each input \(x\) and k sub-key and produces two output words of y, where 1 word is 32 bits. This function is the heart of BCF encryption/decryption. For a note, 1 word is 32 bits.
Table 1. BCF substitution box 1

| Round | Key | Ramdonize |
|-------|-----|-----------|
| 0     | b4  |          |
| 1     | 2b  |          |
| 2     | 54  |          |
| 3     | 3  |          |
| 4     | d1  |          |
| 5     | bd  |          |
| 6     | 75  |          |
| 7     | 34  |          |
| 8     | 84  |          |
| 9     | c4  |          |
| 10    | a  |          |
| 11    | b  |          |
| 12    | c  |          |
| 13    | d  |          |
| 14    | e  |          |

Table 2. BCF substitution box 2

| Round | Key | Ramdonize |
|-------|-----|-----------|
| 0     | 04  |          |
| 1     | 5a  |          |
| 2     | fd  |          |
| 3     | 18  |          |
| 4     | f0  |          |
| 5     | d6  |          |
| 6     | 80  |          |
| 7     | 84  |          |
| 8     | 9f  |          |
| 9     | 8b  |          |
| 10    | d5  |          |
| 11    | b  |          |
| 12    | c  |          |
| 13    | d  |          |
| 14    | e  |          |

Table 3. BCF substitution box 3

| Round | Key | Ramdonize |
|-------|-----|-----------|
| 0     | 0   |          |
| 1     | 12  |          |
| 2     | 58  |          |
| 3     | 9-12|          |
| 4     | 17-18|        |

Table 4. BCF substitution box 4

| Round | Key | Ramdonize |
|-------|-----|-----------|
| 0     | 1   |          |
| 1     | 2   |          |
| 2     | 3   |          |
| 3     | 4   |          |
| 4     | 5   |          |
| 5     | 6   |          |
| 6     | 7   |          |
| 7     | 8   |          |
| 8     | 9   |          |
| 9     | 10  |          |
| 10    | 11  |          |
| 11    | 12  |          |
| 12    | 13  |          |
| 13    | 14  |          |

Table 5. The algorithm for selecting SBox

| Round | Key | Ramdonize |
|-------|-----|-----------|
| 1-4   | K_A |          |
| 5-8   | K_A |          |
| 9-12  | K_A |          |
| 13-18 | K_A |          |
P is the product between the matrix M and the input x with an aim to obtain optimal diffusion.

\[ b = P(x) = M \cdot x \]  

The input is \( x = \{ x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7 \} \) and the output is \( P(x) = b = \{ b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7 \} \). BCF uses the same irreducible polynomials used by AES, \( m(x) = x^8 + x^4 + x^3 + x + 1 \).

2.3. BCF Key Expansion

The BCF keyschedule (key expansion) algorithm has a key input of 128 bits (16 bytes or 4 words), and performs a Key Expansion to generate some sub-keys. The Key Expansion produces a total of 17 sub-keys, 15 sub-keys for the regular round (K0, K1, ..., K14) and 4 sub-keys for whitening keys (KW1, KW2, KW3, KW4). If the primary key is 192 bits or 256 bits, then we perform XOR operation between the left and the right side of the primary key so that it still generates a key of 128 bits to be included in the key schedule.

At the beginning of the key schedule, the intermediate keys: K0, K1, ..., K14 are generated. From these intermediate keys, all sub-keys required for the encryption and decryption processes are generated. Figure 2 shows the beginning of the key expansion to generate K0, K1, K2, and K3.

![Fig. 2 BCF key expansion](image)

Table 6 describes the complete key expansion process. The table is connected to the figure 2 as key expansion process.

| Key | Key Function |
|-----|--------------|
| KE  | (K0 \( \bigcup \) K1) \( \oplus \) KE |
| KF  | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| KG  | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| Kw1 | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| Kw2 | KE \( \oplus \) KE |
| K1  | KE \( \oplus \) KE |
| K2  | KE \( \oplus \) KE |
| K3  | (K0 \( \bigcup \) 0x0000000000000000) \( \oplus \) KE |
| K4  | KE \( \oplus \) KE |
| K5  | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| K6  | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| K7  | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| K8  | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| K9  | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| K10 | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| K11 | KE \( \oplus \) KE |
| K12 | (K0 \( \bigcup \) KE) \( \oplus \) KE |
| K13 | KE \( \oplus \) KE |
| K14 | KE \( \oplus \) KE |
| K15 | KE \( \oplus \) KE |
| K16 | KE \( \oplus \) KE |
| K17 | KE \( \oplus \) KE |
| K18 | KE \( \oplus \) KE |
| KW1 | KE \( \oplus \) KE |
| KW2 | KE \( \oplus \) KE |
| KW3 | KE \( \oplus \) KE |
| KW4 | KE \( \oplus \) KE |

2.4. BCF Decryption

The BCF decryption (Figure 3) procedure can be performed in the same way as encryption, but with the sub-key order reversed. More details are shown in the following pseudo code.

\[
X_R = X_R \oplus KW_4 \\
X_L = X_L \oplus KW_3 \\
X_L = X_L \oplus F0(K_{18}, X_R) \\
X_R = X_L \\
X_R = X_L \oplus F0(K_{17}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_{16}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_{15}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_{14}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_{13}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_{12}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_{11}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_{10}, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_9, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_8, X_R) \\
X_L = X_R \\
X_R = X_L \oplus F0(K_7, X_R) \\
X_L = X_R \\]
X_R = X_L \wedge F_0(K_7, X_R) \\
X_L = X_R \\
X_R = X_L \wedge F_0(K_6, X_R) \\
X_L = X_R \\
X_R = X_L \wedge F_0(K_5, X_R) \\
X_L = X_R \\
X_R = X_L \wedge F_0(K_4, X_R) \\
X_L = X_R \\
X_R = X_L \wedge F_0(K_3, X_R) \\
X_L = X_R \\
X_R = X_L \wedge F_0(K_2, X_R) \\
X_L = X_R \\
X_R = X_L \wedge F_0(K_1, X_R) \\
X_L = X_R \\
X_R = X_L \wedge KW_2 \\
X_L = X_L \wedge KW_1

Fig. 3 BCF decryption

2.5. BCF Core IP Design

IP BCF core design is implemented in Verilog HDL language, top down method. The design begins by defining the system, making the architecture first, and designing the supporting modules. The IP BCF core symbol and pin out are shown in Figure 4. Table 7 describes the function of each pin.

Figure 5 shows the general architecture of BCF. Key_len pin out functions to set the number of rounds and the number of rounds depends on the size of the key. For a key with a size of 128 bits, 192 bits, 256 bits, it takes 15, 16, and 18 rounds, respectively. The decrypt pin out determines whether BCF_Core will encrypt or decrypt. The core of the BCF Engine contains encryption, decryption and keyschedule. Figure 6 illustrates the core algorithm for the BCF engine.

| Port Name | Port Width (bit) | Direction | Description |
|-----------|-----------------|-----------|-------------|
| clk       | 1               | Input     | System clock |
| rst       | 1               | Input     | 0: Reset BCF Core |
| sel       | 1               | Input     | 1: Active BCF Core |
| decrypt   | 1               | Input     | 0: Decryption |
|           |                 |           | 1: Encryption |
| key_len   | 2               | Input     | 0 : 128 bits |
|           |                 |           | 01 : 192 bits |
|           |                 |           | 10 : 256 bits |
| key       | 256             | Input     | BCF key |
| din       | 128             | Input     | Data input |
| dout      | 128             | Output    | Data output |
| done      | 1               | output    | 0 : BCF Process Done |
|           |                 |           | 1 : BCF Processing |

Fig. 4 Core pin out

This core architecture was used interchangeably for randomizing and key schedule, resulting in large latency. For the encryption and decryption process, the BCF engine required 316 clocks.

Figure 7 shows the BCF timing diagram. The system was controlled by clk.

To implement BCF, we needed a FSM (Finite State Machine). The BCF FSM is shown in Figure 8.
Figure 9 and Figure 10 illustrate the keys schedule (key expansion) architecture and the FO module, respectively. One Core FO was used interchangeably in encryption, decryption, and key schedule. The advantage of this design is to use a small area but it has the disadvantage of having a large latency.

The further explanation of FSM in Figure 8 and FO Core in Figure 9 is described in more detailed in Table 14 and Table 15 (appendix).

Subbyte BCF operations were implemented using LUTs for the ease of design and minimization of critical paths [4].

The description of FO module pins is described in Table 16 (appendix).

The search for the substitution box number was obtained from this equation:

\[
si = ((K >> ((2^i) + (8^r))) & 8'h03) \uparrow ((K >> (8^r + 2)) & 8'h03) \uparrow ((K >> (8^r + 4)) & 8'h03) \uparrow ((K >> (8^r + 6)) & 8'h03)) & zero
\]  

(4)
The MixColumn operation used a systolic array architecture (Figure 11). The architecture used only eight processing elements in MixColumn and one processing element in Subbyte processing. The tradeoff of this architecture was the latency from one clock to eight clocks.

Figure 12 shows the Processing Element Design. If input data is \( x \) and multiplier number in mixcolumn is \( M \), then the result of polynomial multiplication between \( x \) and \( M \) is \( b \).

The design of processing elements is implemented using the architecture as shown in Figure 13.

Table 8 shows some descriptions of the Processing Element pinouts. Pinout R contains the result of polynomial multiplication between data from pinout A and B. Xtime algorithm was used for polynomial multiplication in mixcolumn operation. For efficiency in Mixcolumn operation, shift and xor operation was applied [3]. Figure 14 depicts the architecture of xtime algorithm implemented in mixcolumn.

Table 9 shows some descriptions of the XTime pinouts. Xtime architecture used in this design had one input data and eight output data. It was purposely to enable the polynomial multiplication to be completed in one clock.
Table 9. Xtime pinout description

| Port Name | Data Length (bit) | Direction | Description                             |
|-----------|-------------------|-----------|-----------------------------------------|
| d         | 8                 | Input     | Data input xtime function.              |
| x1        | 8                 | Output    | Data output for d multiple by 1<sub>i</sub> |
| x3        | 8                 | Output    | Data output for d multiple by 3<sub>i</sub> |
| x4        | 8                 | Output    | Data output for d multiple by 4<sub>i</sub> |
| x5        | 8                 | Output    | Data output for d multiple by 5<sub>i</sub> |
| x6        | 8                 | Output    | Data output for d multiple by 6<sub>i</sub> |
| x7        | 8                 | Output    | Data output for d multiple by 7<sub>i</sub> |
| x8        | 8                 | Output    | Data output for d multiple by 8<sub>i</sub> |
| xB        | 8                 | Output    | Data output for d multiple by B<sub>i</sub> |

Table 10. Synthesis comparation

| Algorithm  | Logic Element | Register | FMax (MHz) | Key length (bit) | Throughput (Bps) at 50 MHz |
|------------|--------------|----------|------------|------------------|-----------------------------|
| BCFV1      | 5790         | 396      | 54.7       | 256              | 7594937                     |
| SASEBO AES | 5122         | 396      | 84.84      | 128              | 123076923                   |
| SASEBO Camellia | 4251 | 397      | 59.36      | 128              | 55172413                    |
| SASEBO TDEA | 1017        | 256      | 74.4       | 192              | 38461538                    |

2.6. BCF Integration to FPGA Atera DE II

DE2-115 is a development board with the main component in the form of Altera Cyclone® IV 4CE115 FPGA. Soft core NIOS processor can be implemented on FPGA. NIOS is a soft core 32-bit RISC Microprocessor. In this paper, we used 50 MHz frequency on BCF. The BCF module was wrapped with the Avalon interconnect interface. Figure 15 shows the block diagram of the NIOS interface.

![Fig. 15 BCF - NIOS interface](image)

To access IP BCF, NIOS write / read registers in Table 11. The functionality test of IP BCF was carried out by comparing the computing results of BCF IP with program that run on a computer. Figure 16 and 17 show the result.

Based on the above test, the ciphertext and plaintext values generated by the IP Core BCF were found similar with those generated by the C program running on the computer. This means that the implementation of BCF on BCF has been functionally successful.
One way to measure the BCF performance is by comparing the hardware and software implementation. If the speed of the hardware far exceeds the speed of the software, the hardware implementation can be stated to be successful. The measurement results are presented in Table 12 showing that the computation time for BCF software implementation depends on the processor architecture. Hardware BCF Engine can speed up BCF compute 488-2847 times compared to software, dependent on processor architecture and BCF key length.

### Table 11. Register for software

| Address   | W/R | Description                               |
|-----------|-----|-------------------------------------------|
| 0x81080   | W   | Write to register Key on bit 0 to 31      |
| 0x81081   | W   | Write to register Key on bit 32 to 63     |
| 0x81082   | W   | Write to register Key on bit 64 to 95     |
| 0x81083   | W   | Write to register Key on bit 96 to 127    |
| 0x81084   | W   | Write to register Key on bit 128 to 159   |
| 0x81085   | W   | Write to register Key on bit 160 to 191   |
| 0x81086   | W   | Write to register Key on bit 192 to 223   |
| 0x81087   | W   | Write to register Key on bit 224 to 255   |
| 0x81088   | W   | Write to register din on bit 0 to 31      |
| 0x81089   | W   | Write to register din on bit 32 to 63     |
| 0x8108A   | W   | Write to register din on bit 64 to 95     |
| 0x8108B   | W   | Write to register din on bit 96 to 127    |
| 0x8108C   | W   | Write to register key_len 0: 128 bit      |
| 0x8108D   | W   | Write to register key_len 1: 192 bit      |
| 0x8108E   | W   | Write to register key_len 2: 256 bit      |
| 0x8108F   | W   | Write to register decrypt 1: Mode encryption |
| 0x81090   | R   | 0: BCF not done                           |
| 0x81091   | R   | 1: BCF done                               |
| 0x81092   | R   | Read register dout on bit 0 to 31         |
| 0x81093   | R   | Read register dout on bit 32 to 63        |
| 0x81094   | R   | Read register dout on bit 64 to 95        |
| 0x81095   | R   | Read register dout on bit 96 to 127       |
| 0x81096   | W   | Reset BCF Engine                          |

The speed of BCF and AES was measured, and the results of the comparison are shown in Table 13 informing the BCF Engine was 44 times faster than the AES hardware accelerator, where the devices operated at a clock of 50MHz. From this data, BCF Engine is suitable to be implemented in devices with small computing resources such as IoT, where these devices require a low clock for power saving, but require a high level of security for sending data to the internet [17].

### Table 12. Software and hardware running time comparison

| Version        | NIOS II Average execution time (μS) |
|----------------|-------------------------------------|
| NIOS II        | 256 bit 141664 290 135977 255 127704 239 |
| BCF Engine     | 192 bit 172815 273 159609 261 153015 253 |
| NIOS II        | 128 bit 760116 267 701536 267 672323 267 |

### Table 13. Comparision of Block Cipher 128-bit implementation on FPGA

| Design       | Algorithm | Execution time (μS) |
|--------------|-----------|---------------------|
| Sideris et al [16] | AES | 10414               |
| This work    | BCF       | 239                 |

### 3. Conclusion

This paper describes the BCF encryption algorithm, the algorithm implementation on the Altera DE2-115 FPGA and its performance. On Altera DE2-115 boards, hardware implementations were found 488-2847x faster than software implementations, dependent on processor architecture and BCF key length. BCF also has a high speed to be implemented on devices with small resources such as IoT. For further research, we will perform a Correlation Power Analysis (CPA) attack on this proposed BCF device. The attack will be based on the previous paper [2].

### References

1. Y. Kurniawan. *The BCF block cipher design*, Technical Report, Institut Teknologi Bandung, Indonesia, 2018.
2. Ma’mari, Y. Kurniawan and S. Sutikno, *Implementation of BC3 encryption algorithm on FPGA Zynq-7000*, Int. Symp. Electronics Smart Devices, Yogyakarta, Indonesia, 2017, pp. 329-334.
3. Hua Li and Z. Friggstad, *An efficient architecture for the AES mix columns operation*, IEEE Int. Symp. Circuits Syst., Kobe, 2005, pp. 4637-4640.
4. S. Ghaznavi, C. Gebotys and R. Elbaz, *Efficient technique for the FPGA implementation of the AES mixcolumns transformation*, Int. Conf. Reconfigurable Computing FPGAs, Quintana Roo, 2009, pp. 219-224.
5. A. O. Adebayo, M. S. Chaubey, and L. P. Lumbu, *Industry 4.0: The fourth industrial revolution and how it relates to the application of internet of things(IoT)*, J. Multidisciplinary Eng. Sci. Studies 5 (2019) 2477-2482.
6. I. A. Landge and H. Satopy, *Secured IoT through hashing using MD5*, Fourth Int. Conf. Adv. Electric., AEEICB, Chennai, 2018, pp. 1-5.
7. S. D. Putra, M. Yudhijawara, Y. Kurniawan, S. Sutikno and A. S. Ahmad, *Security analysis of BC3 algorithm for differential power
analysis attack, Int. Symp. Electronics Smart Devices (ISESD), Yogyakarta, 2017, pp. 341-345.

8. P. C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In: Koblitz N. (eds) Advances in Cryptology — CRYPTO ’96. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 1996.

9. Altera, DE2-115 cyclone II development board user manual, Altera, 2010

10. A. Laboratory, Cryptographic hardware project: IP core, http://www.aoki.ecei.tohoku.ac.jp/crypto/web/cores.html, 2019.

11. W. Stallings. Cryptography and network security principles and practices(4th ed). New Jersey: Prentice Hall, 2005.

12. P. S. L. M. Barreto and V. Rijmen. The Khazad Legacy-level Block Cipher, NESSIE, 2001.

13. A. E. Standard., Federal Information Processing Standards Publication, FIPS PUB 197, 2001.

14. K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, T. Tokita. Specification of camellia — a 128-bit block cipher. NTT and Mitsubishi Electric Corporation 2000-2001, 2001, pp. 1-35.

15. B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, Twofish: a 128-bit block cipher. Proceedings first AES cand. conf., 1998.

16. A. Sideris, T. Sanida and M. Dasygenis, Hardware acceleration of the AES algorithm using Nios-II processor, Panhellenic Conf. Electronics Telecommunications, Volos, Greece, 2019, pp. 1-5.

17. R. Mahmoud, T. Yousaf, F. Aloul and I. Zualkernan, Internet of things (IoT) security: Current status, challenges and prospective measures, 10th Int. Conf. Internet Tech. Secured Transactions, London, 2015, pp. 336-341.

Appendix

Table 14. FSM state description

| State | Description |
|-------|-------------|
| S0    | Reset state |
| S1    | Pre-Whitening |
| S2    | round = 1 |
| S3    | round = 15 |
| S4    | round = 16 |
| S5    | round = 18 |
| S6    | Conditional state |
| S7    | \( \text{dinR} = \text{KM}, \text{dinL} = \text{KL} \) |
| S8    | \( \text{dinR} = \text{XRreg}, \text{dinL} = \text{XLreg} \) |
| S9    | Wait until F0 done, then \( \text{XRreg} = \text{XR}, \text{XLreg} = \text{XL} \) |
| S10   | Input data to KReg and KBreg register |
| S11   | Key Schedule finished |
| S12   | Process(Key Schedule) not finished, go to S6 state, ronde = ronde + 1 |
| S13   | Conditional state |
| S14   | Process (encryption or decryption) not finished, go to S6 state, ronde = ronde + 1 |
| S15   | Process (Key Schedule or encryption or decryption) finished, goto S6 state, if Key Schedule finished and KeyS = 1, then KeyS = 0 and go to state S1. |

Table 15. “continued”. FO core pinout description

| Port Name | Data Length (bit) | Direction | Description |
|-----------|------------------|-----------|-------------|
| din       | 128              | Input     | Data input, plain text or cipher text. |

Table 15. “continued”. FO core pinout description

| Port Name | Data Length (bit) | Direction | Description |
|-----------|------------------|-----------|-------------|
| dinR      | 64               | Input     | Data input right |
| dinL      | 64               | Input     | Data input left |
| KS        | 64               | Input     | Subkey |
| XR        | 64               | Output    | Data output right |
| XL        | 64               | Output    | Data output left |
| KS1…KS18  | 64               | Input     | Subkey for each round |

| State | Description |
|-------|-------------|
| KS1   | KeyS = 1, then KS1 = C3. Decrypt = 1, then KS1 = K1. |
| KS2   | KeyS = 1, then KS2 = C4. Decrypt = 1, then KS2 = K2. |
| KS3   | KeyS = 1, then KS3 = K1R. Decrypt = 1, then KS3 = K3. |
| KS4   | KeyS = 1, then KS4 = K1L. Decrypt = 1, then KS4 = K4. |
| KS5   | KeyS = 1, then KS5 = C1. Decrypt = 1, then KS5 = K5. |
| KS6   | KeyS = 1, then KS6 = C2. Decrypt = 1, then KS6 = K6. |
| Key_len = 0 & decrypt = 0, then KS6 = K10. | Key_len = 2 & decrypt = 0, then KS12 = K7. |
|---|---|
| Key_len = 1 & decrypt = 0, then KS6 = K11. | KeyS = 1, then KS13 = 0. |
| Key_len = 2 & decrypt = 0, then KS6 = K13. | Decrypt = 1, then KS13 = K13. |

| KS7 | 64 | Input |
|---|---|---|
| KeyS = 1, then KS7 = K7. | KeyS = 1, then KS14 = K14. |
| Decrypt = 1, then KS7 = K7. | Key_len = 0 & decrypt = 0, then KS14 = K2. |
| Key_len = 0 & decrypt = 0, then KS7 = K9. | Decrypt = 1, then KS14 = K14. |
| Key_len = 1 & decrypt = 0, then KS7 = K10. | Key_len = 1 & decrypt = 0, then KS13 = K4. |
| Key_len = 2 & decrypt = 0, then KS7 = K12. | Key_len = 2 & decrypt = 0, then KS13 = K6. |

| KS8 | 64 | Input |
|---|---|---|
| KeyS = 1, then KS8 = K8. | KeyS = 1, then KS15 = K15. |
| Decrypt = 1, then KS8 = K8. | Decrypt = 1, then KS15 = K15. |
| Key_len = 0 & decrypt = 0, then KS8 = K8. | Key_len = 0 & decrypt = 0, then KS15 = K1. |
| Key_len = 1 & decrypt = 0, then KS8 = K9. | Key_len = 1 & decrypt = 0, then KS14 = K5. |
| Key_len = 2 & decrypt = 0, then KS8 = K11. | Key_len = 1 & decrypt = 0, then KS15 = K1. |

| KS9 | 64 | Input |
|---|---|---|
| KeyS = 1, then KS9 = K9. | KeyS = 1, then KS16 = K16. |
| Decrypt = 1, then KS9 = K9. | Decrypt = 1, then KS16 = K16. |
| Key_len = 0 & decrypt = 0, then KS9 = K7. | Key_len = 0 & decrypt = 0, then KS15 = K1. |
| Key_len = 1 & decrypt = 0, then KS9 = K8. | Key_len = 1 & decrypt = 0, then KS16 = K1. |
| Key_len = 2 & decrypt = 0, then KS9 = K10. | Key_len = 2 & decrypt = 0, then KS16 = K3. |

| KS10 | 64 | Input |
|---|---|---|
| KeyS = 1, then KS10 = K10. | KeyS = 1, then KS17 = K17. |
| Decrypt = 1, then KS10 = K10. | Decrypt = 1, then KS17 = K17. |
| Key_len = 0 & decrypt = 0, then KS10 = K6. | Key_len = 0 & decrypt = 0, then KS17 = K2. |
| Key_len = 1 & decrypt = 0, then KS10 = K7. | Key_len = 2 & decrypt = 0, then KS17 = K2. |
| Key_len = 2 & decrypt = 0, then KS10 = K9. | KeyS = 1, then KS18 = K18. |

| KS11 | 64 | Input |
|---|---|---|
| KeyS = 1, then KS11 = K11. | Key_len = 2 & decrypt = 0, then KS18 = K1. |
| Decrypt = 1, then KS11 = K11. | Key_len = 2 & decrypt = 0, then KS18 = K1. |
| Key_len = 0 & decrypt = 0, then KS11 = K5. | Subkey untuk randomizing SBox. |
| Key_len = 1 & decrypt = 0, then KS11 = K5. | KM, KL, |
| Key_len = 1 & decrypt = 0, then KS11 = K6. | KJ, KK, KL, |
| Key_len = 2 & decrypt = 0, then KS11 = K8. | KWA, |
| Key_len = 2 & decrypt = 0, then KS11 = K6. | KWB, |
| cntF0 | 4 | Input |
|---|---|---|
| Data counter ronde BCF. |

| KS12 | 64 | Input |
|---|---|---|
| KeyS = 1, then KS12 = K12. | KeyS = 1, then KS18 = K18. |
| Decrypt = 1, then KS12 = K12. | KeyS = 1, then KS18 = K18. |
| Key_len = 0 & decrypt = 0, then KS12 = K4. | Key_len = 2 & decrypt = 0, then KS18 = K1. |
| Key_len = 1 & decrypt = 0, then KS12 = K5. | Key_len = 2 & decrypt = 0, then KS18 = K1. |
| KeyS1 | 1 | Input |
|---|---|---|
| 0: Encrypt/Decrypt |
| 1: Key Scheduling |

| dF0 | 1 | Input |
|---|---|---|
| 0: Output Pre-Whitening |
| 1: Output F0 |

| KeyS1 | 1 | Input |
|---|---|---|
| 0: XRRReg, XLReg |
| 1: KM, KL |
Table 16. F0 module pinout description

| Port Name | Data Length (bit) | Direction | Description                      |
|-----------|------------------|-----------|----------------------------------|
| dinR      | 64               | Input     | Data input F0 function           |
| dinL      | 64               | Input     | Data input F0 function           |
| Ks        | 64               | Input     | Subkey                           |
| S…S7     | 3                | Input     | Sbox number used                 |
| cnt       | 4                | Input     | Data counter.                    |
| XL        | 64               | Output    | Data output F0 function          |
| XR        | 64               | Output    | Data output F0 function          |