Design Flow of Blowfish Symmetric-Key Block Cipher on FPGA

Piotr Palka, Rafael A. Perez, Tianyang Fang and Jafar Saniie

Embedded Computing and Signal Processing (ECASP) Research Laboratory (http://ecasp.ece.iit.edu/)
Department of Electrical and Computer Engineering
Illinois Institute of Technology, Chicago, IL, U.S.A.

Abstract— **Cryptography is an essential topic for modern digital systems. In the past two decades, FPGA-based encryptions has proved to be effective. Through this project, we explore a cryptographic application of FPGAs by implementing the Blowfish symmetric-key block cipher on a Xilinx ZedBoard. The motivation behind this is to explore how FPGAs can be used to accelerate the performance of a symmetric-key block cipher. Our goal is to compare the performance of a hardware implementation of Blowfish to a software implementation. Our Blowfish block cipher is implemented in VHDL and is functionally verified against a Python reference model. The design as a whole supports user input of various keys for key initialization along with full encryption and decryption functionality. Furthermore, the design was successfully synthesized and implemented on the ZedBoard FPGA. A performance evaluation was conducted between the hardware implementation and a Blowfish library running in Python by comparing the time it takes to perform the key initialization and encrypt/decrypt operation.**

Keywords— Block cipher, FPGA, cryptography, blowfish.

I. Introduction

Cryptography is defined as the practice and study of techniques for securing communication. This is typically achieved through various encryption algorithms which define protocols for parties to communicate in the presence of an adversary. Encryption algorithms include classical examples such as Caesar and substitution ciphers and modern examples such as Data Encryption Standard (DES) and Advanced Encryption Standard (AES). The main functionality of these algorithms is to take a plaintext, encrypt it in a way that cannot be understood by adversaries, and have a method of decryption for the receiver to retrieve the original plaintext. With this being said, modern-day cryptography extends beyond just encryption and decryption and addresses aspects such as authentication, confidentiality, data integrity, and non-repudiation. Modern-day cryptography includes hash functions, message digests, certificates and signatures, public-key exchange, and a plethora of other techniques which make up various cryptosystems. Applications of modern-day cryptography include and are not limited to: e-commerce, credit card transactions, cryptocurrency, data storage, and secure communication.

Field Programmable Gate Arrays (FPGAs) on the other hand are defined as an integrated circuits that can be reconfigured for various applications by the user. FPGAs are characterized by their high parallel processing capabilities, low power consumption, and flexibility due to their ability to be reconfigured. With this being said, FPGAs are typically utilized as hardware accelerators. They can be paired with a computing resource such as a server and configured to execute a specific function. The advantage of this is that greater efficiency and performance are achieved when running a specific function on hardware compared to software. Software implementations execute on a traditional server where the CPU needs to be shared with other applications. In hardware implementations, the resources on the FPGA are dedicated to that specific application. In addition to this, there is less overhead on the entire system in a hardware implementation compared to a software implementation.

Due to the advantages mentioned above, there is increasing interest in the topic of FPGA-based cryptography systems. Paper [1] implements a pipelined version of AES on an FPGA. Paper [2-7] all concentrated on the implementation of the Blowfish Cipher, although they focused on different aspects that impact the performance: paper [2] explored the feasibility of the design in an IoT application; paper [3] provided a pipelined implementation of the algorithm which improved the performance of Blowfish on FPGAs; paper [4][5] discussed the power consumption and throughput of the proposed designs, while paper [5] also claimed that the low complexity of Blowfish, compared to the standardized AES, means it's better suited for FPGA implementations; paper [6] proposed an enhanced Blowfish algorithm, which provided the potentials of improving the block cipher system on the theoretical side; paper [7] designed a hybrid cryptosystem consisting of both Blowfish and RSA on an FPGA. Through this project, we explore cryptographic applications of FPGAs by implementing the Blowfish symmetric-key block cipher on a Xilinx ZedBoard FPGA. Our goal for this project is to explore the differences in performance of a hardware implementation of the Blowfish cipher compared to a software implementation. Our choice of the Blowfish cipher over something more traditional such as AES revolves around its uniqueness and flexibility. Blowfish is a valid block cipher that is easy to make sense of and is highly effective. It was invented by Bruce Schneier to replace DES in 1993 and is not subject to any patents meaning anyone is free to use it. Blowfish is also flexible in the sense that it supports varying key lengths from 32-bits up to 448-bits. An FPGA implementation for this makes perfect sense as the design can be adapted to support varying key lengths. In addition to this, Blowfish also has a small memory footprint (~4KB) meaning it can be a viable encryption solution for smaller embedded applications.

In the next chapter, we will first introduce the Blowfish Block Cipher algorithm, which includes the algorithm's key initialization, encryption, and decryption; Chapter III talks about the implementation of Blowfish on FPGA in a modularized nature; Chapter IV discusses the performance of the proposed implementation and compares it with running the algorithm with CPUs; finally, Chapter V concludes the paper.

II. BLOWFISH BLOCK CIPHER

Blowfish is a symmetric-key block cipher that operates on blocks of 64-bits at a time. It was invented in 1993 by Bruce Schneier to replace DES. Blowfish operates with variable key lengths which are chosen by the user and can be anywhere from 32-bits to 448-bits. This allows the user to experiment with a tradeoff between speed (shorter keys) and security (longer keys). Blowfish is based on a 16-round Feistel network that utilizes four different s-boxes within the F-function for each round of encryption and decryption.

A. Key Initialization

Prior to encrypting or decrypting, Blowfish goes through a key initialization process which encrypts the initial values in each P-array and S-box with the keys supplied by the user. This key initialization process is summarized as the following:

1) A P-array holding 18, 32-bit entries and four S-boxes each holding 256, 32-bit entries are initialized with the hexadecimal values of PI.

2) The keys supplied by the user can range from 32-bits to 448-bits in length. The user can supply anywhere from 1 to 14 of these keys in a chosen length.

3) A bitwise XOR operation is performed on each P-array entry with the keys supplied by the user. Keys are reused once the last element in the P-array is reached:

$$P_1 \oplus K_1, P_2 \oplus K_2, \dots, P_{14} \oplus K_{14}, P_{15} \oplus K_1, \dots$$

4) A process of encrypting each P-array entry and each S-box entry is started. This is done by encrypting a 64-bit block of all zeros. The result is assigned to P-array entries 1 and 2. Encryption continues for each P-array and S-box entry by encrypting the output of the previous encryption:

$$P_1, P_2 = E[0];\ P_3, P_4 = E[P_1 \parallel P_2];\ P_5, P_6 = E[P_3 \parallel P_4] \dots$$

$$S_{1,1}, S_{1,2} = E[P_{17} \parallel P_{18}];\ S_{1,3}, S_{1,4} = E[S_{1,1} \parallel S_{1,2}] \dots$$

B. Encryption and Decryption

In total around 4KB of memory is required to store all the entries within the P-array and all the entries from the four S-boxes. The Feistel structure along with the functionality of the F-function is summarized in Figures 1 and 2. Encryption occurs through a 16-round Feistel structure with the addition of a final output whitening round which reverses the last swap and performs an XOR operation to get the output. Decryption is easily done through Blowfish by reversing the order in which the P-array entries are used. For encryption, entries are used in the order from 1 to 18 and for decryption, entries are used in the order 18 down to 1. Each round of the Feistel network utilizes an F-function. The F-function takes a 32-bit input and splits that up into 8-bit portions which are used as inputs for each S-box. Outputs from the S-boxes and either added together with modulo 2^{32} addition or bitwise XORed.

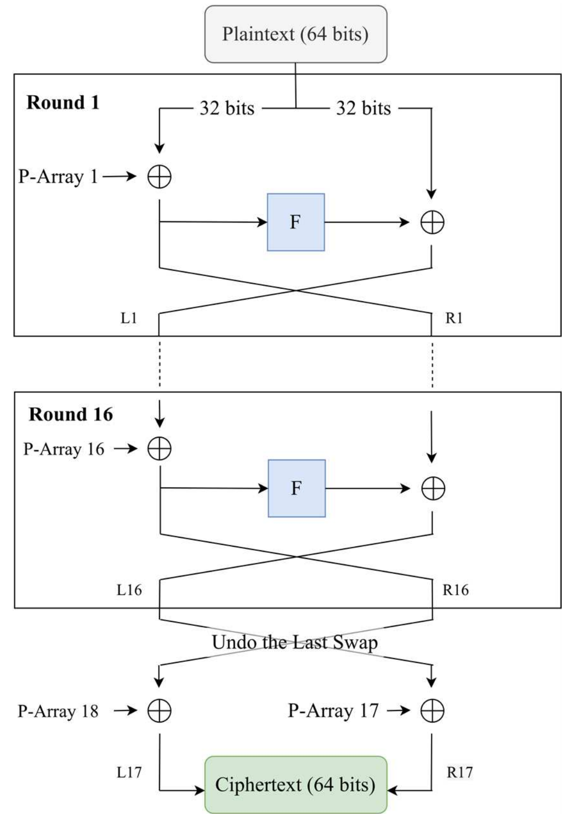

Figure 1. Sixteen Round Feistel Structure of Blowfish (encryption)

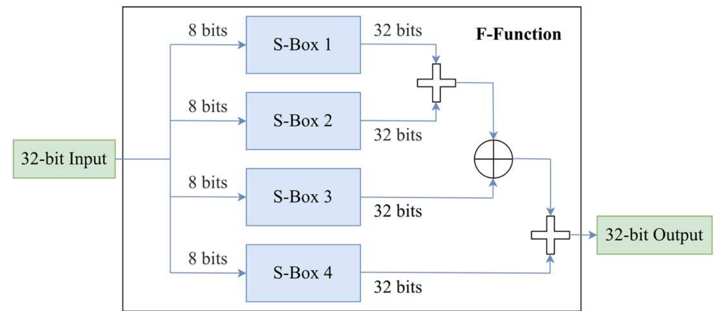

Figure 2. F-function within a single round of Blowfish [8]

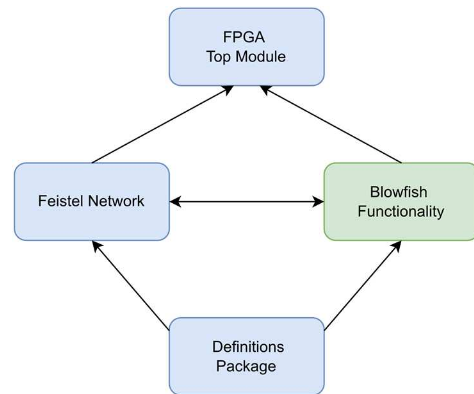

Figure 3. Blowfish FPGA implementation flow chart

III. Blowfish FPGA Implementation

This section gives an overview of the system design of the Blowfish FPGA implementation, which is visualized in figure 3. The implementation is composed of four main VHDL modules which are all described in the subsections which follow.

A. *Definitions Package*

The definitions package defines various constants and functions used within the Blowfish implementation. These constants include the number of rounds within the Feistel network, the number of entries within the P-array and S-boxes, key length, etc. The purpose of this package is to have a centralized file where the constants can be changed to modify the implementation of Blowfish. In addition to this, the definitions package also defines types for the P-array and S-boxes and defines various functions used within the implementation. These functions include initializing the P-array and S-box entries with the hexadecimal values of PI and converting the output of Blowfish into an ASCII format which could be displayed on the OLED of the ZedBoard.

B. *Feistel Network*

The Feistel Network VHDL module implements the functionality of the 16-round Feistel network of Blowfish described in the "background" section above. This module supports both encryption and decryption by utilizing an input signal which is specified by the user. This module also implements the F-function which is utilized in each round of encryption or decryption.

C. *Blowfish Functionality*

The Blowfish Functionality module facilitates the key initialization process and the main encryption/decryption operations based on control signals passed to the module. Key initialization is started with the user asserting the "key_init" signal. The user specifies how many keys (1 - 14) they are supplying the module one by one. Key initialization begins by reading in these keys and populating an array of them. Once all keys have been taken in, the key-encryption process begins as described in the "background" section above. With the key initialization process completed, all P-array and S-box entries are properly encrypted and Blowfish is ready to encrypt or decrypt user input. The encryption/decryption operation starts with the user setting the 64-bit input, setting the "enc_dec" signal to specify encryption or decryption operation, and then asserting the "exec" signal to start the process. The "ready" signal will be asserted once the process of encrypting is finished. The output can then be read from the output line of the module.

D. *FPGA Top Module*

The FPGA Top Module ties together all the previously discussed modules into a single module which gets synthesized and implemented on the FPGA. It utilizes an OLED controller module to be able to display the output to the OLED display on the ZedBoard and a clock divider IP core to be able to set a frequency for the design utilizing the onboard clock. The entire design is implemented on the programmable logic of the ZedBoard. The input switches are utilized for user input, where each switch corresponds to a set of preprogrammed inputs which are entered as soon as the switch is toggled. The output is then displayed on the OLED display on the ZedBoard.

E. *Python Reference Implementation*

Lastly, to verify the functionality of the VHDL implementation another Blowfish cipher was implemented entirely within Python. The functionality of this Python implementation was verified against a Python Blowfish library. The reason for this custom implementation was to be able to display every internal operation and compare it against the VHDL implementation. The P-array and S-box entries were initialized to the hexadecimal values of PI and encrypted with the same keys used in our VHDL implementation. Through this, the proper functionality of the VHDL implementation was verified as every internal signal matched the internal operations of the Python model.

IV. Experiment Results and Evaluation

Verifying the Blowfish implementation began in software via a behavioral simulation in Vivado. The waveforms and internal signals from this simulation were compared to the reference Python implementation. The hardware implementation was verified to be correct as the output along with all the internal signals matched those of the Python reference implementation. A test bench was created to start the key initialization process, encrypt user input, and decrypt user input. The keys used for this simulation were: *FEDCBA98* and *76543210*, both being 32-bits in length. Once the key initialization process was completed, a sequence of encryption and decryption operations were performed. The 64-bit input: *0123456789ABCDEF* was first encrypted. The expected output of this operation is *0ACEAB0FC6A0A28D*. Following this, the output was then decrypted to retrieve the original input. Both encryption and decryption operations take 18 clock cycles. This makes sense in our implementation as one clock cycle is needed to load the input, 16 rounds are needed within the Feistel structure, and one final round is needed for the output whitening stage. After 18 clock cycles, the output of encryption or decryption is visible on the output line. This output then gets picked up a clock cycle later. In addition to this, we also found that the entire key initialization process (from asserting the key-init signal to the time the ready signal rises) takes a total of 9398 clock cycles. These results are in line with the 521 total encryption operations which are required during the key initialization process [8]. It takes 9 encryptions to initialize the entire P-array of 18 entries and 128 encryptions for each of the four S-boxes. Each encryption operation takes 18 clock cycles so this equates to a total of 9378 clock cycles. Before the encryption step, there are 18 total XOR operations which when added to the total number of clock cycles results in 9396. The last two clock cycles are accounted for as overhead with setting the key init signal at the beginning and then asserting the ready signal at the end.

Following this, the design was synthesized and implemented on the actual FPGA. The FPGA Top Module was utilized to interact with our design through the hardware. Interaction with the design begins by pressing the "start" button. This begins the key initialization process and completes when the "key-init-done" LED turns on. At this point, the "ready" LED should also be on, signifying the board is ready to encrypt or decrypt user input. Input is passed to the board via switches. Each switch is preprogrammed to write data to the "data-in" line, specify

encryption or decryption operation, and start the specified operation. The output can be viewed on the OLED display by refreshing the screen. On the OLED display itself, the first line corresponds to the user input, the second corresponds to the operation (encryption or decryption), and the last line corresponds to the output. The hardware implementation produces the same results as the behavioral simulations which were verified to be correct.

We found that our implementation operates successfully at a maximum frequency of 75MHz. Anything higher than this results in our design not meeting timing constraints. Based on our behavioral simulation our encrypt/decrypt operation takes 18 clock cycles and the key initialization process takes 9398. We use this information to conduct a performance evaluation comparing our hardware implementation against a Blowfish module running in Python on an AMD Ryzen 5 3600 at 3.6 GHz, an Intel i7 6700k at 4.5 GHz, and an Intel i5 1035G7 at 1.2 GHz. The Python Blowfish implementation takes an average of 10,000 runs for each operation. Random inputs (keys and user inputs) are generated between runs to avoid any caching. Table I below summarizes our findings.

Table I. Performance evaluation comparing key initialization

	Key Initialization	Encrypt / Decrypt
AMD 3600	33.64e-4 sec	65.82e-7 sec
Intel 6700k	31.92e-4 sec	64.28e-7 sec
Intel 1035G7	40.24e-4 sec	76.43e-7 sec
ZedBoard	1.253e-4 sec	2.53e-7 sec

From the results in Table I, it can be seen that the hardware implementation experiences better performance compared to the software implementation. Comparing our hardware implementation to each of the software runs, we observe a significant improvement in time compared to all three processors running the software implementation. There is a significant increase in the key init times and the encrypt/decrypt times in the Python implementation which is most likely caused by the extra overhead which is present in the software. It's interesting to view the encryption/decryption operation latencies as bytes encrypted/decrypted per second. The result of this is seen below.

Table II. Performance comparison by encrypt / decrypt bandwidths

	Encrypt / Decrypt
AMD 3600	1.22 MB/s
Intel 6700k	1.24 MB/s
Intel 1035G7	1.05 MB/s
ZedBoard	31.6 MB/s

As seen in Table II, all the results are not particularly exceptional in terms of encryption/decryption bandwidth. The hardware implementation manages 31.6 MB/s, which would be sufficient to fully encrypt a modern residential household internet connection twice over 1 (and that's ignoring mode-of-operation overhead). This bandwidth figure is more than enough for most embedded applications, but likely not sufficient for data-intensive server tasks. Part of the reason for this lackluster performance is the 64-bit block size - if a bigger block size could be used without dramatically affecting the maximum frequency, the bandwidth could be improved.

Looking back at operation latencies, comparing the hardware implementation to the software implementation in terms of percentage decrease results in the following table.

Table III. Percent decrease in execution time

ZedBoard Vs.	% Decrease (key init)	% Decrease (encrypt/decrypt)
AMD 3600	96.28%	96.16%
Intel 6700k	96.07%	96.06%
Intel 1035G7	96.89%	96.69%

From these results, we can conclude that the hardware implementation of Blowfish experiences an overall 96% decrease in execution time when compared to a software implementation running in Python.

V. CONCLUSION

Through this project, we successfully implemented the Blowfish symmetric-key block cipher on a Xilinx ZedBoard FPGA. Our implementation was verified to be functionally correct against a reference Python implementation that we designed. Our implementation takes 18 clock cycles for each encryption or decryption operation and 9398 clock cycles for the key initialization process. These results are in line with reference implementations we found online.

We were successful to synthesize and implement the design on the actual hardware. Through our top module, the user interacts with the design through the onboard switches, LEDs, and OLED display. We found that our design performs at a maximum frequency of 75MHz. We conducted a performance evaluation comparing our hardware implementation to a Python Blowfish module. We ran this Python module on three different systems and calculated the average time per key init and encrypt/decrypt operation.

We conclude that the hardware implementation experiences a significant improvement in performance over the software implementation. We observe that the increased times on the 1 Standard connections from providers like Spectrum start at 12.5 MB/s software implementation are due to the extra overhead which is associated with running on software. Overall, we observed a 96% increase in performance in our hardware implementation when comparing it to software.

REFERENCES

[1] A. P. Anusha Naidu and P. K. Joshi, "FPGA implementation of fully pipelined Advanced Encryption Standard," 2015 International Conference on Communications and Signal Processing (ICCSP), 2015, pp. 0649-0653, doi: 10.1109/ICCSP.2015.7322568.

[2] K. N. Prasetyo, Y. Purwanto and D. Darlis, "An implementation of data encryption for Internet of Things using blowfish algorithm on FPGA," 2014 2nd International Conference on Information and Communication Technology (ICoICT), 2014, pp. 75-79, doi: 10.1109/ICoICT.2014.6914043.

[3] S. R. Chatterjee, S. Majumder, B. Pramanik and M. Chakraborty, "FPGA Implementation of Pipelined Blowfish Algorithm," 2014 Fifth International Symposium on Electronic System Design, 2014, pp. 208-209, doi: 10.1109/ISED.2014.51.

[4] S. B. Nalawade and D. H. Gawali, "Design and implementation of blowfish algorithm using reconfigurable platform," 2017 International Conference on Recent Innovations in Signal processing and Embedded Systems (RISE), 2017, pp. 479-484, doi: 10.1109/RISE.2017.8378204.

[5] R. Ahmad[1], A. A. Manaf and W. Ismail, "Development of an improved power-throughput Blowfish algorithm on FPGA," 2016 IEEE 12th International Colloquium on Signal Processing & Its Applications (CSPA), 2016, pp. 237-241, doi: 10.1109/CSPA.2016.7515838.

[6] V. Poonia and N. S. Yadav, "Analysis of modified Blowfish algorithm in different cases with various parameters," 2015 International Conference on Advanced Computing and Communication Systems, 2015, pp. 1-5, doi: 10.1109/ICACCS.2015.7324114.

[7] V. P. Bansal and S. Singh, "A hybrid data encryption technique using RSA and Blowfish for cloud computing on FPGAs," 2015 2nd International Conference on Recent Advances in Engineering & Computational Sciences (RAECS), 2015, pp. 1-5, doi: 10.1109/RAECS.2015.7453367.

[8] R. Anusha, M. J. Dileep Kumar, V. S. Shetty and N. Prajwal Hegde, "Symmetric Key Algorithm in Computer security: A Review," 2020 4th International Conference on Electronics, Communication and Aerospace Technology (ICECA), 2020, pp. 765-769, doi: 10.1109/ICECA49313.2020.9297547.