Implementing RC6 Cipher Algorithm using VHDL language (Spring 2021)

Eduardo Espino, Gene Drumheller, Ruben Rangel, Mohamed El-Hadedy
ECE-Department, College of Engineering, California State Polytechnic University, Pomona
Email: { ejespino, Grdrumheller, rubenc1, **mealy }** @cpp.edu

***Abstract*—This report describes the background, implementation, and the results of using the RC6 cipher algorithm using the VHDL hardware description language. The goal of the project is to input texts and have them encrypted, and then decrypted back to its original form. RC6 was designed to meet the Advanced Encryption Standards (AES) competition and has reached the top five finalists, thus it is believed to be a reliable and secure encryption process. Using a simulation, we determined the RC6 to be working as intended.**

I. Introduction

THE Rivest Cipher 6 (RC6) algorithm is a system key block cipher designed by Ron Rivest, Matt Robshaw, Ray Sidney, and Yiqun Lisa Yin, first published in 1998. The RC6 was derived from the RC5 algorithm to meet the requirements of the Advanced Encryption Standards (AES) competition, and to increase security and performance. In short, the goal of RC6 is to provide security, be simple, and offer good performance. RC5 and RC6 algorithms are very similar in structure, using data-dependent rotations, modular additions, and XOR operations, and can be seen as interweaving two parallel RC5 encryption processes at once. This form of structure is known as the generalized Feistel cipher (figure 1), also known as the Luby-Rackoff block cipher. A Feistel network uses a round function, which is a function which takes two inputs, a data block, a subkey, and then returns one output that is the same size as the data block. How a round works in RC6 is that half of its data is updated by its other half, and the two are then swapped. Although no practical attack on the RC5 has been found, RC6 was developed to thwart a theoretical attack discovered during a study, based on the fact RC5's rotation does not depend on all of the bits in the register. The proper RC6 has a block size of 128 bits (although block size of 32 bits and 64 bits is possible) and supports different key sizes from 128 bits to up to 2040 bits, and by default have 20 rounds. Although RC6 met the requirements of AES, and became one of the five finalists of the AES competition, it was never selected as a standard, ultimately losing to the Rijndael algorithm. RC6 was proprietary and was patented by RSA Security, however, the patent expired between 2015 and 2017.

In this project, Xilinx's Vivado and VHDL hardware description language code are used to implement the RC6 algorithm into FPGA, which will allow us to determine and evaluate the performance and obtain any novel information of the RC6 cipher.

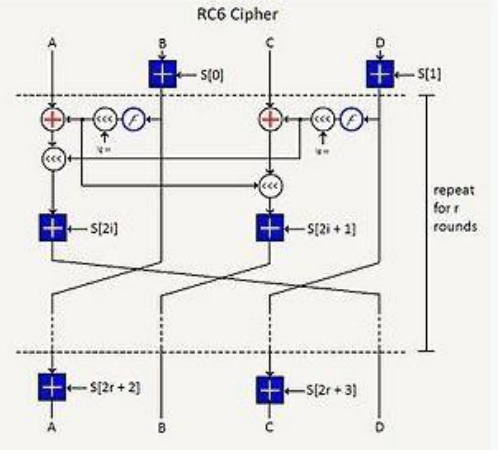

Fig. 1 RC6 Feistel Cipher

II. RELATED WORK

The RC5 was intentionally designed to be really simple to invite analysis on security provided by the extensive use of data dependent rotations, meaning it was relatively a fast operation and required less memory for execution. The RC5 was designed by Ron Rivest and was first published in 1994. As mentioned earlier, RC5 and RC6 have a similar structure, using a variable block size of 32, 64, or 128 bits with key sizes 0 to 2040 bits, and rounds from 0 to 255. All the variable numbers and sizes were implemented to create flexibility. To use the RC5 algorithm, the user inputs text block size, number of rounds, and the key, which must be within its corresponding sizes. Once the values are decided, the values will remain the same during the execution of the cryptographic algorithm.

For an example of a project using the RC6 algorithm, Sudheer Reddy Enugu of National Institute of Technology, Rourkela, India, has implemented it into a Field Programmable Gate Arrays (FPGA) for IPSec protocol using the VHDL hardware description language. IPSec is a framework for security that operates on the network layer by extending the IP packet header, and the purpose of the IPSec protocol is to secure the data while traveling through the network. The RC6 was used to encrypt and decrypt data as different subsystems had to communicate with each other to achieve the final product, including the FPGA, the PC, and a microcontroller. For the results, it was concluded that RC6 is a secure, compact, and a simple block cipher, with good performance and flexibility.

III. Math and Implementation

A more accurate title for RC6 would be specified as RC6-w/r/b, where w stands for word size, r stands for nonnegative number of rounds, and b denotes the length of encryption in bytes. When RC6 was submitted to AES, w was at 32 and r at 20, and b as 16, 24, or 32-byte keys. Typically, RC6 implies the same w and r unless specified otherwise.

$a + b$	integer addition modulo 2^w
$a - b$	integer subtraction modulo 2^w
$a \oplus b$	bitwise exclusive-or of w-bit words
$a \times b$	integer multiplication modulo 2^w
$a \ll b$	rotate the w-bit word a to the left by the amount given by the least significant $\lg w$ bits of b
$a \gg b$	rotate the w-bit word a to the right by the amount given by the least significant $\lg w$ bits of b

Fig. 2 Basic Math Operations of RC6

RC6 works with four w-bit registered A, B, C, and D using the following six basic operations (figure 2). A, B, C, and D will contain the initial input and the output at the end of the encryption. The theoretical basic procedure of the encryption (figure 3) and decryption (figure 4) are shown.

Encryption with RC6-$w/r/b$

Input: Plaintext stored in four w-bit input registers A, B, C, D
Number r of rounds
w-bit round keys $S[0, \ldots, 2r+3]$

Output: Ciphertext stored in A, B, C, D

Procedure:
$B = B + S[0]$
$D = D + S[1]$
for $i = 1$ **to** r **do**
{
$t = (B \times (2B+1)) \lll \lg w$
$u = (D \times (2D+1)) \lll \lg w$
$A = ((A \oplus t) \lll u) + S[2i]$
$C = ((C \oplus u) \lll t) + S[2i+1]$
$(A, B, C, D) = (B, C, D, A)$
}
$A = A + S[2r+2]$
$C = C + S[2r+3]$

Fig. 3 RC6 Encryption Procedure

Decryption with RC6-$w/r/b$

Input: Ciphertext stored in four w-bit input registers A, B, C, D
Number r of rounds
w-bit round keys $S[0, \ldots, 2r+3]$

Output: Plaintext stored in A, B, C, D

Procedure:
$C = C - S[2r+3]$
$A = A - S[2r+2]$
for $i = r$ **downto** 1 **do**
{
$(A, B, C, D) = (D, A, B, C)$
$u = (D \times (2D+1)) \lll \lg w$
$t = (B \times (2B+1)) \lll \lg w$
$C = ((C - S[2i+1]) \ggg t) \oplus u$
$A = ((A - S[2i]) \ggg u) \oplus t$
}
$D = D - S[1]$
$B = B - S[0]$

Fig. 4 RC6 Decryption Procedure

IV. Design

The design of RC6 was made with security, simplicity, and performance in mind, based on the experiences gained using the RC5. During the preceding development, the decision to expand to four 32-bits of registers were made for performance reasons. Next, the decision to make the quadratic formula f(x)=x(2x+1)(mod2^w) was made later. This quadratic function change is aimed towards providing a faster rate of diffusion, thus improving the chance that simple differentials will use the rotation amounts much sooner than RC5 would. Simplicity was kept in mind because when a cipher is simple, it can be analyzed widely by other researchers. The simplicity allows experimentation by both simple and complex processes, which is useful to evaluate the security of the algorithm. As RC6 was selected as the top five contested for AES, the encryption is quite secure, with no major issues found. Such testing was made by basic cryptanalytic attacks, linear and differential cryptanalysis attacks, and other attacks.

V. Simulation

We began the RC6 implementation with a key expansion algorithm repository we found on GitHub (https://github.com/EasonNYC/FPGA-RC5/blob/master/key_exp.vhd). This repo implements this pseudocode:

```
Input: array L[0 ... c-1] of input key words
Output: array S[0 ... 43] of round key words --S[0 ... (2r + 3)]

Procedure:
S[0] = 0xB7E15163
for i = 1 to 43 do S[i] = S[i-1] + 0x9E3779B9
A = B = i = j = 0
for k = 1 to 132 do
        { A = S[i] = (S[i] + A + B) <<< 3
          B = L[j] = (L[j] + A + B) <<< (A + B)
          i = (i + 1) mod 44 -- 2r + =4
          j = (j + 1) mod c }
```

Fig. 5 RC6 Repo Pseudocode

The key expansion algorithm is used to expand the user-supplied key to fill an expanded array, S. S resembles an array of t random binary words. The user must supply a key of b bytes, where 0 ≤ b ≤ 255, and from which (2r+4) words are derived and stored in a round key array S. Zero bytes are appended to give the key length equal to a "non-zero integral number". The key bytes are then loaded in little-endian order into an array L of size c; when b = 0, c = 1 and L[0] = 0. The (2r+4) derived words are stored in array S for decryption or encryption.

For the encryption part of the implementation, we began by assigning the inputs and outputs as shown in Figure 5.

Next, we declared the key expansion component and all the necessary signals (Figure 6).

Fig. 5 Inputs/Outputs for encryption

Fig. 6 Key expansion declaration and signals

We, then, instantiated the key expansion and rearranged the bytes in the four registers per this statement in *The RC6 Block Cipher*, "The first byte of plaintext or ciphertext is placed in the least-significant byte of A; the last byte of plaintext or ciphertext is placed into the most-significant byte of D." (Figure 7)

Fig. 7 Key expansion and byte rearrangement

In this next section, we finally began to implement the RC6 encryption algorithm by starting the loop. The inputs initially take the first couple values of the S array and then once inside the for loop, they take the values of each register, A, B, C, and D for the previous iteration as inputs. Then, we assigned 2B and 2D to the appropriate signals and then added one to each value and assigned them to their own variables; this was primarily done for readability. Those values were finally multiplied by the B & D values, respectively. (Figure 8).

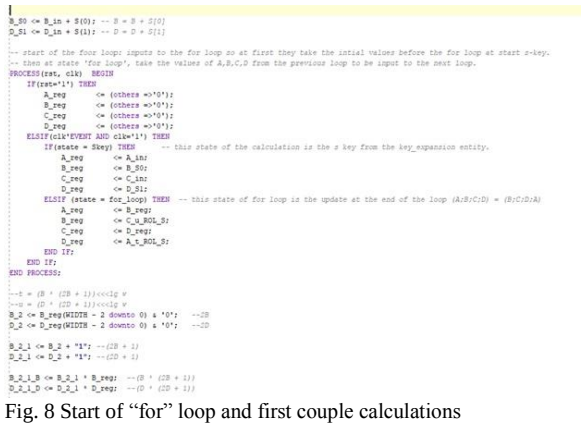

Fig. 8 Start of "for" loop and first couple calculations

The next step was to the modulus of the previous signals and, because VHDL does not allow us to use the modulus operator when synthesizing, we used an alternate method, which is basically to take only 32 bits of the previous signals (B_2_1_B & D_2_1_D). The last part of the first two calculations was simply to rotate signals to obtain *t* and *u*. Next, we began to implement the A & C calculations by using the XOR operator on the A register and C register on t and u, respectively. We then rotated left by the least significant values of t & u for each value of A and C from 1 to 31, similar to what was done in the key expansion. (Figure 9)

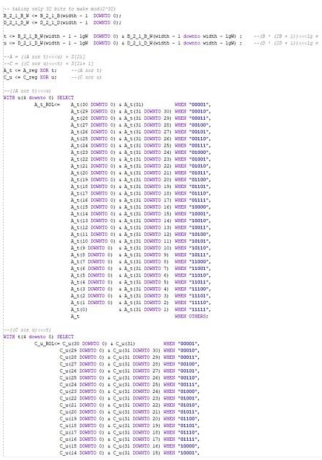

Fig. 9 Mod and rotations

We then added S[2i] to the rotated A signal and S[2i + 1] to the rotated C signal, which we obtained in the previous step, and assigned the parallel assignment at the end of the food loop. Lastly, we added A to S[2r + 2] and C to S[2r + 3] to obtain the final A and C values in the algorithm.

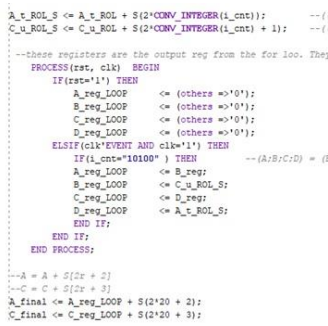

Fig. 10 Parallel assignment and final A & C values

The decryption part of the algorithm was implemented similarly, except basically it was the reverse as shown above.

To simulate and test each part of the algorithm, the following test cases were used (Figure 11):

```
plaintext 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
user key 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
ciphertext 3A 96 F9 C7 F6 75 5C FE 46 F0 0E 3D CD 5D 2A 3C

plaintext 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
user key 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ciphertext 8f c3 a5 36 56 b1 f7 78 c1 29 df 4e 98 48 a4 1e

plaintext 02 13 24 35 46 57 68 79 8a 9b ac bd ce df e0 f1
user key 01 23 45 67 89 ab cd ef 01 12 23 34 45 56 67 78
ciphertext 52 4e 19 2f 47 15 c6 23 1f 51 f6 36 7e a4 3f 18
```

Fig. 11 Test Cases

We had a little difficulty implementing this into a test bench using the standard methods we learned in class, so instead, we learned how to run a TCL script and tested the code using this method. The code produced the expected results as shown in Figure 12 for the encryption part and Figure 13 for the decryption part.

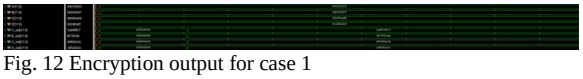

Fig. 12 Encryption output for case 1

Fig. 13 Decryption output for case 1

We attempted to implement this onto the FPGA board, however we could not get it to compile successfully. The code synthesizes fine but when attempting to run the implementation the code failed. The primary error we were receiving was due to overutilization. We tried multiple methods to correct the issue and simplify the code. Unfortunately, none of the fixes we attempted worked and we ran out of time before we could resolve the issue.

VI. Conclusion

The purpose of this project is to understand the theoretical background of the RC6 cipher, and to experiment to determine the actual performance and obtain any novel information by writing a VHDL code and simulating on the test bench. The project utilized Xilinx's Vivado to observe the efficiency of the RC6 algorithm. Our results indicate that the RC6 algorithm works as intended and is a reliable way to encrypt and decrypt the data. Performance is essential to determine so that improvements can be implemented to better conserve and utilize resources, whether it be hardware or software related.

References

[1] RC6: The Simple Cipher, Morgan Monger , CS-627-0001: Cryptography Fall 2004, James Madison University

[2] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 Block Cipher" [1]M.I.T. Laboratory for Computer Science, Cambridge, MA, USA. [2]RSA Laboratories, San Mateo, CA, USA. Version 1.1- Aug. 20, 1998 Accessed: May 142021. [Online]. Available: http://people.csail.mit.edu/rivest/pubs/RRSY98.pdf

[3] S. R. Enugu, "FPGA Implementation of RC6 Algorithm for IPSec protocol," M.S thesis, National Institute of Technology, Rourkela - Dept. Electron. Comms. Eng., 2008. [Online] Available: https://core.ac.uk/download/pdf/53188906.pdf

[4] S. Tawde, "RC5," EDUCBA, 02-Mar-2021. [Online]. Available: https://www.educba.com/rc5/. [Accessed: 14-May-2021].

[5] https://en.wikipedia.org/wiki/RC6

[6] https://en.wikipedia.org/wiki/RC5

[7] https://www.educba.com/rc5/