RFCA: Efficient, Robust and Flexible Cipher Algorithm For FPGA Implementation

Raphael Couturier[1] and Hassan Noura[1]

[1]Université de Franche-Comté, CNRS, institut FEMTO-ST, F-90000 Belfort, France

KEYWORDS

Cryptography, FPGA, Lightweight Cipher Scheme, Security and Performance Analysis.

ABSTRACT

The current Field-Programmable Gate Array (FPGA) implementation of cryptographic algorithms faces performance and security challenges because these algorithms were not originally designed to take FPGA features into account. One significant performance limitation arises from the iteration of a round function for a high round number, given the fixed structures like static substitution and diffusion primitives throughout the process. This paper introduces a new framework for a key-dependent, flexible one-round stream cipher scheme specifically designed to benefit from FPGA features. It is called RFCA. Security and performance analyses validate the effectiveness and robustness of the proposed solution, ensuring the desired cryptographic properties. In comparison with an AES implementation, RFCA is 34 times faster.

INTRODUCTION

Field-Programmable Gate Array (FPGA) is programmable hardware that can be configured to implement digital circuits. FPGAs provide an adaptable hardware platform for designing, implementing and testing digital circuits. The FPGA's flexibility allows designers to balance high-level abstraction with fine-grained control over low-level details. This resulting in novel, efficient hardware solutions optimized for cryptographic algorithms. FPGA can be used to design and implement cryptographic algorithms, such as encryption/hashing, deterministic pseudo-random generator and key generation. These algorithms are often computationally intensive. Improved performance and consequently reduced execution delay and power consumption are two advantages of using FPGAs for cryptographic purposes. These are important considerations for real-time applications and/or devices with limited battery power. Another reason why FPGAs are a preferred option is because of their wide use and the well-established standards in the semiconductor industry, which offer a community of designers and engineers enabling the continuous development and optimization of cryptographic implementations.

However, several difficulties and restrictions appear when using FPGA to implement established cryptographic algorithms Wang et al. (2011) such as AES yaa (2021), including high resource overhead and limited flexibility due to the structure of these algorithms (higher number of rounds) and the fixed input block length such as 128 bits in the case of Advanced Encryption Standard (AES). These algorithms were designed as general solutions and independent of hardware characteristics, which makes them unable to reach the best performance. Therefore, in this work, we design a stream cipher algorithm according to the FPGA characteristics such as parallelism, and flexible precision to optimize performance and resource utilization in the FPGA.

The proposed stream cipher reaches a high level of efficiency, robustness, and flexibility. Efficiency is achieved as the proposed cipher scheme is realized according to FPGA characteristics and by avoiding the limitations of current cryptographic algorithms that require a higher number of rounds as the employed cryptographic primitives are static. This was achieved by employing the dynamic cryptographic approach, where cryptographic primitives are variable and this can prevent cryptanalysis techniques, which are based on the concept of the static key approach. Therefore, the proposed stream cipher requires only one round instead of a higher number of rounds to minimize the computation complexity Noura et al. (2019). Additionally, the proposed stream cipher ensures flexibility as the size of the block stream is variable and depends on the number of words h and the word precision Wp to meet application and/or device needs.

To the best of our knowledge, this research presents a novel dynamic cipher designed for FPGAs, offering both high speed and strong security, making it a valuable advancement in the field.

Organization

The structure of the paper is as follows: In the next section, a description of the proposed stream cipher and its principal components are provided. After this, a comprehensive security analysis is conducted, evaluat-

ing the proposed cipher against essential cryptographic properties and the security level of the proposed cipher against diverse attack techniques. Then, the efficiency of the proposed cipher solution is analyzed compared to AES.

Proposed Stream Cipher Scheme: RFCA

The input message M is divided into nb blocks, denoted as $M = m_1,\ m_2,\ \ldots,\ m_{nb}$, with each block comprising h words. Fortunately, the overhead of the conversion operation delay is eliminated with FPGA since there is no need to perform any conversion operations to word or byte representation. This indicates that a block of input, either plaintext or ciphertext, can be processed at the byte/bit level or with word precision Wp, for instance, a size of 32 or 64 bits. There will be h words in each block, which represents a shared value between the source and destination. This means that the size of each block is $h \times Wp$ bits. In this work, we set h to 4 and the word precision Wp to 64 bits. Hence, the plain or ciphertext blocks have the same representation. Furthermore, XorShift64 is selected as a PRNG due to its low computational complexity. Note that any other secure and efficient PRNG can be used instead of XorShift64.

On the other hand, as depicted in Figure 1, the j^{th} ciphertext block (c_j) is generated by combining the j^{th} plaintext block (m_j) with the j^{th} produced key-stream block V_j. The detailed process is outlined in the listing code 1 and expressed in Eq. 1:

$$c_j = E_K(m_j) = m_j \oplus V_j\ ;\ \ j = 1,\ 2,\ \ldots,\ nb \quad (1)$$

The generated key-stream (Vector V) is updated iteratively: the $(j+1)^{th}$ vector is updated as a function of the j^{th} key-stream vector. The proposed stream cipher comprises one function, which is the Round Function (RF). It consists of iterating two functions in parallel (f and g), as illustrated in Figure 1-a). The output of the f and g functions will be combined for every round function iteration to create a block of keystream V. Furthermore, these functions are recursive, meaning that the $j+1$ iteration's input will be the result of iteration j. The $(j+1)^{th}$ encrypted block c_{j+1} is obtained by combining (XOR) the $(j+1)^{th}$ plaintext block with the $(j+1)^{th}$ key-stream V.

The Round Function (RF)

Then, as previously said, the **Round Function (RF)** is iterated by iterating the recursive functions f and g and mixing their output to generate a keystream block, which yields the j^{th} key-stream block V_j. We suggest two potential functions for f and g in Figure 1-b, respectively, as a proof of concept. For additional information, note that the function f only uses an efficient PRNG (XorShift64), with the current iteration's result serving as the subsequent iteration's input. On the other hand, apart from using PRNG, the function g consists of multiple operations, including substitution, permutation, and non-invertible binary diffusion. Based on randomness analysis, the optimal order of operations in the function g was determined. The function g was constructed using diffusion as its first operation, which involved iterating a PRNG after utilizing a non-invertible binary matrix. Substitution and byte permutation operations, which are based on permutation and substitution tables, come next in the output of the PRNG. These procedures are explained in full below.

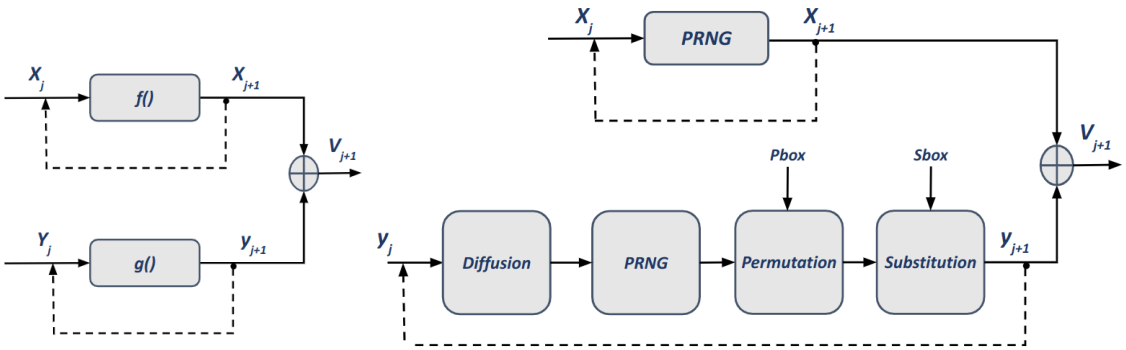
1. ***Iterate PRNGs***: Iterate an efficient PRNG once using the elements as seeds for each element of vector X. Store each output at the corresponding index in vector X, updating X in the process.
2. **Binary Mixing Diffusion Operation**: Combines the word elements of X with the seeds vector V in a non-invertible binary matrix D.

$$Y' = D \odot Y \quad (2)$$

   In fact, this step is added to prohibit recovering the states from any keystream by combining the input seeds of PRNGs in a non-invertible way. Because the binary diffusion matrix is non-invertible, this is made possible.
3. ***Permutation Operation***: Using the generated dynamic permutation table, a byte permutation process is applied for the PRNG's output, which has a length of $\frac{h \times Wp}{8}$ bytes.
4. ***Substitution Operation***: For the permuted diffused block with length $\frac{h \times Wp}{8}$ bytes, a byte substitution process is applied utilizing one or more generated dynamic substitution table(s). This is determined by the configuration. To guarantee that the generated keystream block has a high degree of non-linearity, only one replacement table is needed. The function g, whose result will combine with the function f's output to produce the keystream block, is represented by the output of the substitution process.

The ciphertext $C = c_1||c_2||\ldots||c_{nb}$ is created by encrypting each plaintext block with its matching key-stream block. The jth plaintext block m_j is xored with its matching key-stream word V_j, which is equal to $Y_j \oplus X_j$.

The key-stream blocks generation V_j and dynamic key production are handled by the decryption algorithm using the same procedures. These are then "xored" with the ciphertext blocks c_j to produce the plaintext message block m_j. That is, as stated in Eq. 3: in order to retrieve the j^{th} plain block m'_j, the j^{th} ciphertext block c_j is mixed (XOR) with the j^{th} generated key-stream word V_j.



(a) (b)

Figure 1: General structure of the proposed round function (RF) of the proposed stream cipher for the j^{th} block (iteration) (a) and an example of possible implementation (b)

$$m'_j = E_K(c_j) = c_j \oplus V_j \ , \ j = 1,\ 2,\ \ \ldots,\ nb \quad (3)$$

Next, we provide an example of the use of "XorShift64" as a PRNG for the proof of concept.

Xorshift64 PRNG

"XorShift64" is employed in the proposed stream cipher as it is an efficient pseudo-random generator, which yields an output block of 64 bits, for a word precision of $Wp = 64$. The PRNG is iterated h times, with the result being saved at index w of X and an input seed value $X[w]$ utilized in each iteration of the function f. It is applied recursively, with the current iteration's output serving as the input for the subsequent iteration.

Algorithm 1 provides an explanation of the Linear-Feedback Shift Registers (LFSR) family, of which XorShift is the successor. The latest implementation of XorShift makes it extremely fast since it does not require non-linear operations or too sparse polynomials Panneton and L'ecuyer (2005). Although it is very effective, it does not pass several statistical tests Panneton and L'ecuyer (2005). This problem is addressed in the suggested strategy by using dynamic bit rotation to attain the required level of non-linearity. It should be noted that XorShift can be substituted with any PRNG or a combination of several PRNGs.

The following section presents the FPGA implementation of the proposed stream cipher.

FPGA implementation of RFCA

The suggested approach was created specifically to be used with FPGAs. The Silice framework has been used to develop this method as efficiently as possible sil (2024). The explanation of the implementation follows. The idea behind the implementation can be understood without having to know the Silice language. In our system, 256-bit numbers are changed at every stage. In this work, 4 parts based on 64 bits are used, as the majority of popular algorithms use 64 bits. Thus, the RFCA cipher generates 256 bits at each step.

Algorithm 1 xorshift64 PRNG

Input: 64-bit word of state t
Output: A produced random number word x with 64-bits length

1: **procedure** XORSHIFT64(t)
2: $x \leftarrow t$;
3: $x \leftarrow x \oplus (x >> 12)$;
4: $x \leftarrow x \oplus (x << 25)$;
5: $x \leftarrow x \oplus (x >> 27)$;
6: **return** x;
7: **end procedure**

The first step consists of using the non-invertible with 4 variables (from line 1 to 4). Then the xorshift function is called 4 times (from line 7 to 10). The result of the function is set in temporary variables (this is due to a Silice feature). Then the result of the temporary variables is copied into the rm variables (from line 12 to 15). Next, a 256-bit variable is used. Silice allows one to use a parallel loop. So the next step consists of calling the second xorshift with the *nstate* variable (using temporary variables like previously). It should be noticed that the syntax with $$ comes from the Lua language (line 18 to 21). In the next step, the permutation and the Sbox are used with a parallel for (line 22 to 25). Then a very important step consists of waiting for the next clock (with the ++: symbol line 26). In fact, the need for the next clock is explicit. That means that all the previous instructions are called in only one clock. Because the Sbox is called with a RAM, the read and the write of a RAM require 1 cycle. This is why a new clock is needed. So the next step consists of obtaining the result of the Sbox (line 27 to 29). The Sbox is used with 8 bits, that is why 32

parallel calls are used. The next step is used to update the *state* variable of the Xorshift (line 31 to 34). The last instructions are used to put the result inside the stream variable by xoring the number and rm variables (line 36 to 38).

At the end of the while loop, a new clock is started. That means only 2 clocks are necessary to execute this algorithm with an FPGA (that is powerful enough).

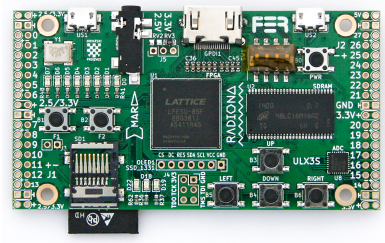


Figure 2: A screenshot of the ULX3S dev board.

Listing 1: Implementation of the RFCA in Silice (only the stream cipher part)

```
1  while (1) {
2      nstate0 = state0 ^ state1;
3      nstate1 = state0 ^ state2;
4      nstate2 = state0 ^ state3;
5      nstate3 = state2 ^ state3;
6
7      (temp0) = XorShift(rm0);
8      (temp1) = XorShift(rm1);
9      (temp2) = XorShift(rm2);
10     (temp3) = XorShift(rm3);
11
12     rm0 = temp0;
13     rm1 = temp1;
14     rm2 = temp2;
15     rm3 = temp3;
16
17     //xorshift call
18     $$for i=0,3 do
19         (temp$i$) = XorShift(nstate$i$);
20         number[$i*64$,64] = temp$i$;
21     $$end
22     // permutation + sbox
23     $$for i=0,31 do
24         sbox$i$.addr=number[perm[$i$],8];
25     $$end
26     ++:
27     $$for i=0,31 do
28    number[$i*8$,8]=sbox$i$.rdata;
29     $$end
30
31     //update of XorShift
32     $$for i=0,3 do
33         state$i$ = number[$i*64$,64];
34     $$end
35
36     $$for i=0,3 do
37         stream[$i*64$,64] = number[$i*64$
               ,64] ^ rm$i$;
38     $$end
39 }
```

The following sections discuss the security and performance analysis needed to evaluate the robustness and efficiency of the proposed stream cipher.

Security Analysis

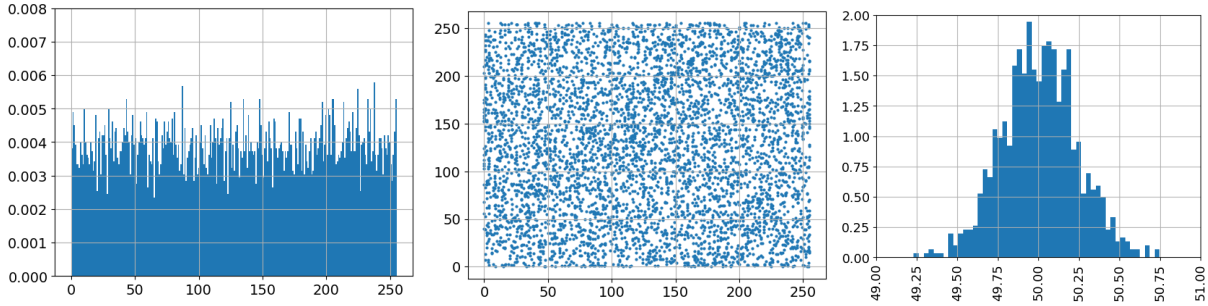
A cryptography solution is designed to withstand many types of analytical attacks, such as differential, statistical, linear, and brute-force attacks Paar and Pelzl (2009); Stallings (2017). This section assesses the suggested stream cipher's resilience to these types of attacks in the worst-case situation.

If the generated keystream exhibits high levels of randomness, unpredictability, and periodicity, statistical attacks can be avoided Stallings (2017). To validate this, the generated keystream is assessed using two benchmark statistical tests: TestU01 L'Ecuyer and Simard (2007) and PractRand Doty-Humphrey (2014). These tests are known for being challenging, but they guarantee that the produced key-streams satisfy the necessary standards of uniformity and randomization, as well as the lack of periodic patterns in the keystream(s) that are produced. These tests' outcomes show that, for every seed tested, the suggested stream cipher reliably passes both the TestU01 and PractRand randomization tests. The histogram of the key-stream with a random session key SK and its associated recurrence in Figure 3-b) are displayed in Figure 3-a), which displays the visual results of the key stream for a size of 10,240 bytes. These findings show that the generated key-stream has a uniform distribution with all symbols having the same probability of occurrence and displays a random recurrence distributed over the entire space. In Noura et al. (2023, 2022a), more implementation details for these tests are given.

The dynamic key-dependent approach Noura et al. (2022b), in which all cryptographic primitives are changed for each new message (or for a series of messages; depends of configuration), is used by the proposed stream cipher to reach high level of security. By generating unique key-streams, this updating approach extends periodicity and improves non-linearity.

Additionally, for 1,000 random dynamic keys with a key-stream length of 10,240 bytes, the percentage difference between created key-streams for a single bit change in the secret session key is evaluated to assess the key sensitivity. Figure 3-c) shows how sensitive the suggested stream cipher is to keys. The key avalanche effect is confirmed by the extremely near difference of the generated ciphertexts (key-streams) to 50%.

Lastly, since the proposed scheme is dynamic and uses a unique key for every new message (or set of messages), the cryptographic primitives will be updated and as a result, different keystream will be produced and lead to obtain different results when encrypting or decrypting the same plaintext/ciphertext within the same session as well as between sessions. Because of its intrinsic variability, the system is protected from analytic attacks that rely on the idea of a static key approach.



(a) (b) (c) KS

Figure 3: The corresponding (a) histogram, and (b) recurrence, of the produced key-stream for a random dynamic key ($h = 16$), and Key sensitivity results (c) for the proposed stream cipher considering 1,000 random keys.

Performance Analysis

The RFCA algorithm has been implemented in several languages: C to verify that the outcomes are the same, Silice for the FPGA, and Practrand and TestU01 test Doty-Humphrey (2014) to verify the key-stream that is successful. The implemented code has been used with the ULX3S development board. This board can be seen in a screenshot in Figure 2. This implementation has been compared to an AES implementation yaa (2021) using that board.

The suggested 256-bit cipher only needs two cycles for its implementation and the frequency is around 85Mhz, so we consider it to be at least 80Mhz. So each clock cycle 128 bits are produced at a frequency of 80Mhz. So the throughput is equal to $128 \times 85 \times 10^6 = 1.024 \times 10^{10}$ bits per second.

In contrast to AES yaa (2021), which produces 128 bits in 42 cycles at 100Mhz, 3 bits are produced per cycle. So the throughput is equal to $3 \times 100 \times 10^6 = 3 \times 10^8$ bits per second.

Consequently, compared to this AES version, the proposed approach is 34 times faster.

Concerning the consumption of resources on the FPGA, our implementation uses 6312 LUTs (Look Up Table) whereas the implementation of yaa (2021) uses 4006 LUTs.

Conclusion & Future Work

In this paper, we present RFCA, a lightweight stream cipher designed to align with FPGA characteristics, making it suitable for real-time applications and/or constrained devices. The approach involves a round function that is lightweight, iterated for only one iteration, and comprises simple operations such as byte substitution, permutation, a lightweight PRNG, and basic logical operations (exclusive or). Furthermore, the proposed cipher incorporates a dynamic key-dependent structure to achieve an optimal balance between security and performance. In comparison to conventional ciphers like AES, the suggested stream cipher significantly reduces both execution time and resource requirements. The effectiveness and resilience of the proposed stream cipher, compared to modern lightweight ciphers and optimized AES implementations, are confirmed through security testing and performance assessments. This work aims to achieve an optimal trade-off between security and performance, aligning with the recent trend of developing lightweight cryptographic algorithms. In future work, we intend to explore possible optimizations for keyed hash functions utilizing FPGA characteristics.

Acknowledgement

This work has been achieved in the frame of the EIPHI Graduate school (contract "ANR-17-EURE-0002").

References

Yet another aes implementation in hardware. `https://github.com/marph91/yaaes`, 2021.

Silice: A language for hardcoding algorithms with pipelines and parallelism into FPGA hardware. `https://github.com/sylefeb/Silice`, 2024.

C. Doty-Humphrey. Practrand, 2014. URL `http://pracrand.sourceforge.net/`.

Pierre L'Ecuyer and Richard J. Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw*, 33(4): 22:1–22:40, 2007. URL `http://doi.acm.org/10.1145/1268776.1268777`.

Hassan Noura, Raphaël Couturier, Congduc Pham, and Ali Chehab. Lightweight stream cipher scheme for resource-constrained iot devices. In *2019 International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–8. IEEE, 2019.

Hassan Noura, Ola Salman, Raphaël Couturier, and Ali Chehab. LESCA: Lightweight stream cipher algorithm for emerging systems. *Ad Hoc Networks*, 138:102999, 2023.

Hassan N Noura, Ola Salman, Raphaël Couturier, and Ali Chehab. LoRCA: Lightweight round block and stream cipher algorithms for iov systems. *Vehicular Communications*, 34:100416, 2022a.

Hassan N Noura, Ola Salman, Raphaël Couturier, and Ali Chehab. A single-pass and one-round message authentication encryption for limited iot devices. *IEEE Internet of Things Journal*, (18):17885–17900, 2022b.

Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009.

François Panneton and Pierre L'ecuyer. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 15(4):346–361, 2005.

William Stallings. *Cryptography and network security: principles and practice*. Pearson Upper Saddle River, NJ, 2017.

Lei Wang, Jiwu Jing, Zongbin Liu, Lingchen Zhang, and Wuqiong Pan. Evaluating optimized implementations of stream cipher zuc algorithm on fpga. In *International Conference on Information and Communications Security*, pages 202–215. Springer, 2011.