electronics

MDPI

Article

Compact Hardware Architectures of Enocoro-128v2 Stream Cipher for Constrained Embedded Devices

Lampros Pyrgas [1,2] and Paris Kitsos [1,2,*]

[1] Electrical and Computer Engineering Department, University of the Peloponnese, Koukouli, 26334 Patras, Greece; pyrgas@isi.gr
[2] Industrial Systems Institute of "Athena" RIC in ICT and Knowledge Technologies, Platani, 26504 Patras, Greece
* Correspondence: kitsos@uop.gr

Received: 11 August 2020; Accepted: 10 September 2020; Published: 14 September 2020

Abstract: Lightweight cryptography is a vital and fast growing field in today's world where billions of constrained devices interact with each other. In this paper, two novel compact architectures of the Enocoro-128v2 stream cipher are presented. The Enocoro-128v2 is part of the ISO/IEC 29192-3 standard. The first architecture has an 8-bit datapath while the second one has a 4-bit datapath. The proposed architectures were implemented on the BASYS3 board (Artix 7 XC7A35T) using the VERILOG hardware description language. The hardware implementation of the proposed 8-bit architecture runs at a 189 MHz clock and reaches a throughput equal to 302 Mbps, while at the same time, it utilizes only 254 Look-up Tables (LUTs) and 330 Flip-flops (FFs). Each round of computations requires 5 clock cycles. The 4-bit implementation has an operating frequency of 204 MHz and reaches a throughput equal to 181 Mbps, with each round requiring 9 clock cycles. The 4-bit implementation utilizes 249 LUTs and 343 FFs. To our knowledge, this is the first time that such implementations of the Enocoro-128v2 are presented. Both implementations utilize a very low number of resources (only 78 FPGA slices are required for the 8-bit architecture and only 83 for the 4-bit one) and the results demonstrate that they are sustainable for area constrained embedded devices.

Keywords: Enocoro-128v2 stream cipher; FPGA implementations; lightweight cryptography; hardware security; constrained embedded devices

1. Introduction

In today's world, wireless communication is an essential part of our life, present in a wide field of our everyday affairs, such as health care, entertainment, or the exchange of information. Personal data are constantly exchanged between devices that are connected with each other, creating a system of interrelated computing devices known as the Internet of things (IoT). In addition, with the development of the fifth generation (5G) technology standard for cellular networks, the amount of the exchanged data, the speed of this exchange, and the number of the simultaneously connected devices are expected to be significantly increased.

As the number of the connected IoT devices rises, their vulnerability to cyber-attacks increases as well, with information being the most common target, through privilege abuse [1,2]. The connected devices, referred as Cyber-Physical Systems (CPS), combine both hardware and software. This combination and the required abstraction between hardware and software leads to safety and security problems and concerns [3]. Therefore, the need for security, through dedicated security modules, is one of the main factors that must be taken into consideration during the design of a device. These devices, however, must be small in size and, due to the fact that they usually run on some type of battery, must have low power dissipation. This leads to resource-constrained devices where the available resources

must be carefully split and distributed between all of the device's modules, leaving a very limited amount of resources for security [4].

Moreover, with additional modules to be constantly added on these devices, the resources that can be utilized for the implementation of cryptographic algorithms (ciphers) are constantly restricted. This problem can be solved through the implementation of lightweight versions of existing ciphers or through the design and the implementation of completely new ones, depending usually on additional requirements such as the cipher's type. Ciphers can be fitted into two big categories: asymmetric and symmetric. Asymmetric ciphers, also known as public key ciphers, use two different keys, a public and a private key, for encryption and decryption. The most important asymmetric ciphers include the Rivest–Shamir–Adleman (RSA) Public-key Cryptosystem, the Digital Signature Algorithm (DSA) and Elliptic Curve Cryptography (ECC) techniques. Symmetric ciphers, also known as private key ciphers, use a common private key, known only to the sender and the receiver, for both encryption and decryption. The Advanced Encryption Standard (AES) and the Data Encryption Standard (DES) are two widely known algorithms of this category.

Symmetric ciphers, depending on the way that encryption and decryption are performed, can be divided to block and stream ciphers. Block ciphers process the incoming data in large blocks of predetermined size. Stream ciphers, on the other hand, encrypt and decrypt the incoming data in significantly smaller parts, usually in one byte parts. Stream ciphers operate as random number generators and generate a pseudo-random output stream. This keystream, as it is widely referred to in literature, is then XORed with the incoming data, in order to encrypt or decrypt the data. The fact that both encryption and decryption are done on each bit independently of the others makes the stream ciphers easier and faster to implement, especially in hardware. It is well known that hardware implementations can offer better security levels as well as better performance regarding power dissipation compared to software implementation [5,6].

In this paper, two compact architectures of Enocoro-128v2 are proposed. They follow the standards and the suggested methods of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) for lightweight cryptography. Lightweight cryptography employs many novel methods that are specifically targeted at constrained devices and can overcome many of the problems of conventional cryptography [7]. The proposed architectures are very efficient for resource-critical devices such as embedded hardware devices, smart cards, or other devices with low power dissipation requirements. In order to reduce the required hardware resources without any loss in efficiency, first, many architectural design optimizations are presented and second, an area-optimized implementation for the cipher's S-box is used.

The first proposed implementation has an 8-bit datapath, a round latency equal to 5 clock cycles, runs on a clock frequency of 189 MHz for both encryption and decryption and achieves a throughput equal to 302 Mbps. At the same time, it requires a very low number of hardware resources, as it utilizes only 254 Look-up Tables (LUTs) and 330 Flip-flops (FFs). The second implementation has a 4-bit datapath, reaches a clock frequency of 204 MHz, and has a latency of 9 clock cycles per round for encryption or decryption, leading to a throughput equal to 181 Mbps, with the utilization of only 249 LUTs and 343 FFs.

The rest of this paper is organized as follows: In Section 2 the specifications of the Enocoro-128v2 stream cipher and all the related work are presented. Section 2 also contains a detailed description of the proposed 8-bit and 4-bit architectures. In Section 3, the results of the proposed architectures' implementations are presented and, in Section 4, are compared with results from implementations of other lightweight ciphers. Finally, the conclusions and directions for future works are presented in Section 5.

2. Materials and Methods

2.1. Specifications of Enocoro-128v2 Stream Cipher

In general, the term Enocoro refers to a family of security algorithms, developed by Hitachi Ltd. in Tokyo, Japan, that are aimed for hardware implementation in constrained environments. The latest member of this family of stream ciphers is called Enocoro-128v2. It was proposed in [8] and [9] as a new concreate 128-bit security algorithm. It is included in the ISO/IEC 29192 [10], an international standard for lightweight cryptography. Specifically, it is included in part three of this standard, which is dedicated on stream ciphers. It is important to note that Enocoro-128v2 can perform encryption and decryption up to 10 times faster than the light-weight implementation of AES-128, while maintaining the same level of security, according to the International Electrotechnical Commission (IEC) [11].

Enocoro-128v2 is a PANAMA-like keystream generator (PKSG) [12], a pseudorandom number generator (PRNG) subclass. It takes two inputs, a 128-bit secret Key and a 64-bit public initialization vector (IV) and produces an 8-bit output on every round. It has a 256-bit storage buffer (consisting of 8-bit subparts, denoted b_0 to b_{31}), a 16-bit internal state (split into two bytes, denoted a_0 and a_1, with the byte a_1 being also the output of the algorithm), and an 8-bit counter, denoted as ctr. The buffer is initially filled with the Key, the IV and six constants that are defined in the algorithm's specifications. Its contents, as well as the contents of the internal state and the ones of the counter, are updated on every computational round. Their initial values are shown in Table 1.

Table 1. Initial values of the Enocoro-128v2 according to the specifications.

ctr	a_0	a_1	b_0 to b_{15}	b_{16} to b_{23}	b_{24}	b_{25}	b_{26}	b_{27}	b_{28}	b_{29}	b_{30}	b_{31}
0x01	0x88	0x4c	KEY	IV	0x66	0xe9	0x4b	0xd4	0xef	0x8a	0x2c	0x3b

The most important part of the algorithm is its two update functions, denoted as ρ and λ. The function ρ consists of two parts, first, an 8-bit substitution box, named S_8, that performs a permutation on the input bits and, second, a linear transformation called L (also referred to as Liner). The S_8-box consists of smaller 4-bit substitution boxes, named S_4, which are connected through a new linear transformation. The computations of Enocoro-128v2 are defined over two finite fields in GF(2^8) and GF(2^4). An element of GF(2^8) is given by the polynomial:

$$\varphi_8 = x^8 + x^4 + x^3 + x^2 + 1 \quad (1)$$

while an element of GF(2^4) is given by the polynomial:

$$\varphi_4 = x^4 + x + 1 \quad (2)$$

On each round, both update functions are executed and the values of both the storage buffer and the internal state are updated. It is important to note that the execution of the algorithm is a two-step processes. The first step, that is called the initialization step and lasts for 96 rounds, and the second step, that lasts until the completion of the algorithm. The only difference between these steps is that during every round of the initialization step the ctr counter's value is incremented by the multiplication by 0x02 in the previously defined finite field GF(2^8), while during the next step the ctr counter's value is permanently set to zero. The complete architecture of Enocoro-128v2 is shown in Figure 1.

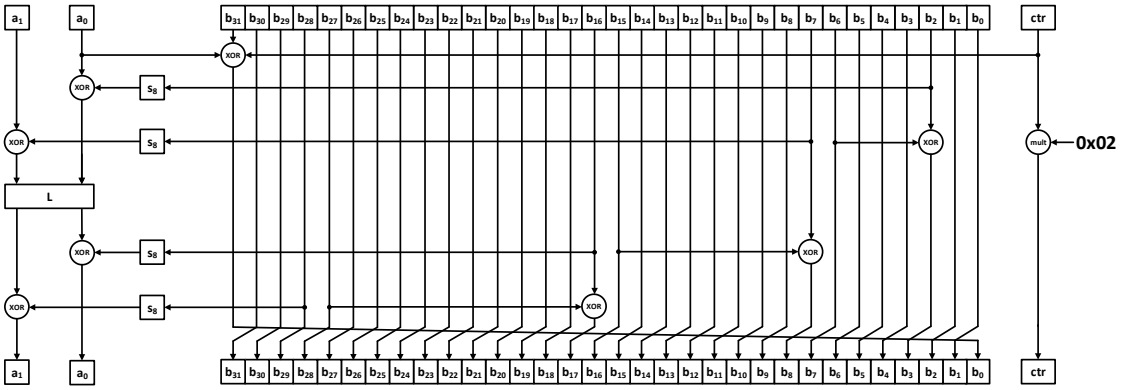

Figure 1. The architecture of Enocoro-128v2 stream cipher according to [9].

2.2. Related Work

The Enocoro-128v2 stream cipher was first proposed in [8] and updated in [9]. In those two works, the specifications of the algorithm were presented. The definition and an architecture for the S_8-box were given as well as the mathematical definition of the linear transformation called L (also referred as Liner) through a 2-by-2 matrix.

Due to the fact that there are not many Enocoro-128v2 FPGA implementations, we also compared our implementations with FPGA implementations of other lightweight stream ciphers. The primary aim of every lightweight stream cipher design is the creation of an architecture that utilizes as few resources as possible while at the same time has a throughput that is high enough for the target application.

In [13] an implementation of DECIM v2 is proposed. This implementation reaches a frequency of 185 MHz and achieves a throughput equal to 46 Mbps, while it utilizes 80 slices and has a T/#Slices factor of 0.58. Additionally, in [13], an implementation of DECIM-128 that can run at a frequency of 174 MHz and has a throughput of 43 Mbps is presented. It utilizes 89 slices and has a T/#Slices factor of 0.49. An implementation of E0 is presented in [14]. It has a maximum frequency of 187 MHz and a maximum throughput equal to 187 Mbps, while it utilizes 140 slices and has a T/#Slices factor of 1.33.

Four implementations of Grain v1 are presented in [13–16]. The first one, in [13], has a frequency of 196 MHz, a throughput of 196 Mbps, it utilizes 44 slices and has a 4.45 T/#Slices factor. The second one, in [14], reaches a frequency of 177 MHz, a throughput of 177 Mbps, it uses 318 slices and has a T/#Slices factor of 0.55. The third one, in [15], has a frequency of 193 MHz, a throughput of 193 Mbps, it utilizes 122 slices and has a 1.58 T/#Slices factor. The fourth one, in [16], reaches a throughput of 105 Mbps, it uses 48 slices and has a T/#Slices factor of 2.19. Two implementations of Grain-128 are presented in [13,17]. The one in [13] has a frequency of 196 MHz, a throughput of 196 Mbps, it uses 50 slices and has a 3.92 T/#Slices factor. The one in [17] has a frequency of 181 MHz, a throughput of 181 Mbps, it uses only 48 slices and has a 3.77 T/#Slices factor.

In [13,14], two MICKEY 2.0 implementations are proposed. The implementation in [13] reaches a frequency of 233 MHz, a throughput of 233 Mbps, it utilizes 115 slices and has a T/#Slices factor of 2.03. The implementation in [14] reaches a frequency of 250 MHz, a throughput of 250 Mbps, it utilizes 98 slices and has a T/#Slices factor of 2.55. In [13,15,17,18] four implementations of MICKEY-128 2.0 are proposed. The one in [13] has a frequency of 223 MHz, a throughput of 223 Mbps, it utilizes 176 slices and has a 1.27 T/#Slices factor. The one in [15] reaches a frequency of 156 MHz, a throughput of 156 Mbps, it uses 261 slices and has a T/#Slices factor of 0.60. The implementation in [17] reaches a maximum frequency of 200 MHz, a throughput of 200 Mbps, it uses 190 slices and has a 1.05 T/#Slices factor. Finally, the one in [18] has a frequency of 170 MHz reaches a throughput of 170 Mbps, it uses 167 slices and has a T/#Slices factor of 1.02.

In [13] and [19], two implementations of Moustique are proposed. The one in [13] has a frequency of 225 MHz, a throughput of 225 Mbps, it uses 278 slices and has a 0.81 T/#Slices factor. The one in [19] has a throughput of 369 Mbps, it uses 252 slices and has a 1.46 T/#Slices factor. In [16] a Mosquito

implementation is presented. It reaches a throughput of 137 Mbps, it uses 298 slices and has a T/#Slices factor of 0.46.

Five implementations of Trivium are presented in [13–17]. The first one, in [13], has a frequency of 240 MHz, a throughput of 240 Mbps, it utilizes 50 slices and has a 4.80 T/#Slices factor. The second one, in [14], reaches a frequency of 326 MHz, a throughput of 326 Mbps, it uses 149 slices and has a T/#Slices factor of 2.18. The third one, in [15], has a frequency of 201 MHz, a throughput of 201 Mbps, it utilizes 188 slices and has a 1.07 T/#Slices factor. The fourth one, in [16], reaches a throughput of 102 Mbps, it uses 40 slices and has a T/#Slices factor of 2.55. The fifth one, in [17], has a frequency of 207 MHz, a throughput of 207 Mbps, it utilizes 41 slices and has a 5.05 T/#Slices factor.

The work in [20] includes two FPGA implementations of the Enocoro-128v2 stream cipher that were optimized based on the throughput/slice (T/#Slices) metric. The first achieves a maximum frequency of 118 MHz while it utilizes 292 slices and has a T/#Slices factor of 0.40. The second reaches a maximum frequency of 149 MHz, requires 442 slices, and has a T/#Slices factor of 0.33.

For the 8-bit architecture of the Enocoro-128v2 that is presented in our paper, we followed the architecture that is presented in [8] and [9] for the S_8-box. However, for our 4-bit Enocoro-128v2 architecture we designed and implemented a novel architecture for the S_8-box that has a 4-bit datapath. For both architectures, we designed and implemented new architectures for the Liner subpart. Our primary aim was to take advantage of the symmetry in the underling matrix and reduce the elements that are needed for the arithmetic calculations. While the elemental data size of the Enocoro-128v2 is a byte, we designed a Liner with a 4-bit datapath for the 4-bit architecture that performs correct calculations while utilizing minimal resources. In addition, we designed a new architecture for the multiplication by 0x02 that has a 4-bit datapath that, again, requires minimal resources. Finally, for both the architectures, we optimized the control logic and the connections between the architecture's subparts in order to keep the resource usage to a minimum.

2.3. Proposed 8-Bit Architecture of Enocoro-128v2 Stream Cipher

The proposed 8-bit architecture, for the Enocoro-128v2 stream cipher, is shown in Figure 2.

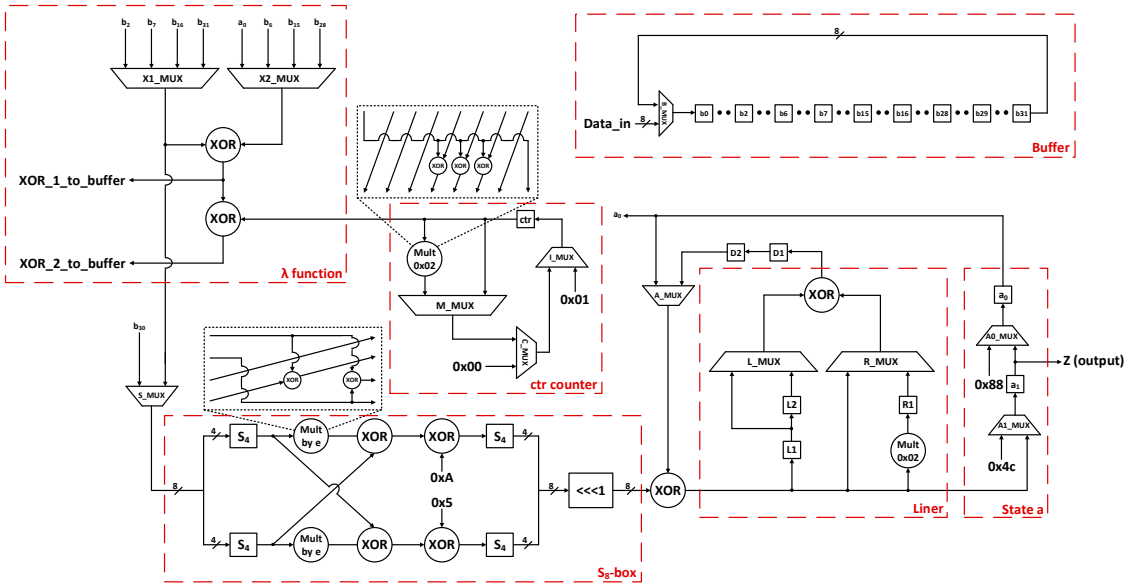

Figure 2. The proposed 8-bit architecture of Enocoro-128v2 stream cipher.

Responsible for the correct operation of the circuit is the Control subsystem, which generates the control signals for the multiplexers. This is done primarily by two counters, one that counts how many rounds have been completed and one that counts how many clock cycles have passed during the current round. The completion of a round requires 5 clock cycles.

In order to store the values that are computed on each round, a storage scheme that follows the algorithm's specifications is required. We opted for a dynamic storage scheme, a buffer named b, instead of a static one. On each round a byte-wise right rotation of the stored values is required. Instead of performing this rotation on a single clock cycle, we perform rotations on each clock cycle in a way that leads to the same result. Specifically, we rotate the values that are stored in the storage buffer by 13 bytes to the right on each of the round's 5 clock cycles. This way the buffer's contents end up in the necessary byte-wise right rotation at the start of the next round. We made this choice, of constantly changing the contents of the buffer, in order to increase the security of our architecture against side channel attacks (mainly Power and Electromagnetic analysis attacks), as measurements that can provide any helpful information for an attacker are more difficult to be obtained.

In addition to the buffer b, the architecture contains three more core components: an internal state named a and two functions named λ and ρ, respectively. The internal state consists of two bytes, a_0 and a_1, that are updated on every round. The byte a_1 is the output of the algorithm. The proposed architecture for the two functions will be described in the following paragraphs.

Prior to the first round of computations, an initialization step is necessary. Following the algorithm's specifications, the registers b_{24} to b_{31} are initialized with the specified constants, the registers b_{16} to b_{23} with the public initialization vector IV and the registers b_0 to b_{15} with the secret input Key. The input Key and the IV are fed to the algorithm byte by byte. After all the above values are stored in the buffer b, in their corresponding positions, the first round of the algorithm can begin.

2.3.1. Function λ of the 8-Bit Architecture

The λ function is responsible for the computation of four XORings, according to the algorithm's specifications in [9]. These XORings are computed, in sequential clock cycles, using the same XOR gate for the first three (e.g., for the input pairs [b_2, b_6], [b_7, b_{15}], [b_{16}, b_{28}] in the initial round). For the fourth XORing an additional XOR gate is required because the result of the XORing of a_0 with the rightmost byte of the buffer (b_{31} in the initial state) must be XORed with the value of the ctr counter. All the final results from the XORings are fed back and stored in the appropriate registers of buffer b. The ctr counter's size is one byte and its initial value is 0x01. The ctr counter's value is incremented by the multiplication by 0x02. The result of this multiplication is stored in the corresponding register in the final clock cycle of the current round, in order to the ctr counter has the correct value at the beginning of the next round. After the completion of 96 rounds, the value 0x00 is stored in the ctr counter and remains unchanged thereafter.

2.3.2. Update Function ρ of the 8-Bit Architecture

The update function ρ consists of two parts, the S_8-box part, implemented according to [9], and the Linear Transformation (Liner) part. The output of the S_8-box for each input byte is produced in the same clock cycle. The input byte is split into two 4-bit parts. Each part is fed into a S_4-box. Each of the two outputs is used in two ways: it is multiplied by the coefficient e = 0x04 (which is defined over the chosen finite field in GF(2^4)) and is also fed into an XOR gate. The gate's second input is the result of the other's S_4-box multiplication by e (the circuit for the multiplication by e is shown in Figure 2). The results of the XOR gates are then fed into two new XOR gates along with the coefficients 0xA and 0x5, respectively. The two outputs are fed into two S_4-boxes. Finally, the two results are concatenated, and the new byte is rotated by one bit to the left and then driven to the output. The output of the S_8-box is then XORed with the appropriate state byte (a_0 or a_1) and is fed to the Liner. This procedure continues for the next bytes, according to the algorithm's specifications.

The Liner has a latency of one clock cycle and its operation lasts for three clocks cycles. In the first clock cycle, the first byte (B1) is fed into the Liner and is stored in register L1. In the second clock cycle, the second byte (B2) is fed into the Liner. Through the left multiplexer, B1 is driven to the XOR gate, while through the right multiplexer B2 is also driven to the XOR gate. The two bytes are XORed and the result is driven to the output. In the same clock cycle, B1 is also moved from L1 and is stored in

register L2. B2 is also multiplied by 0x02 (which is defined over the chosen finite field in GF(2^8)) and the result is stored in register R1. The circuit for the multiplication is shown in Figure 2. Finally, in the third clock cycle, the contents of the registers L2 and R1 are driven to the XOR gate by the left and right multiplexer, respectively. The operation's result is driven to the output. At the appropriate clock cycles, the Liner's output is then XORed with the corresponding S_8-box output according to the algorithm's specifications. The results which correspond to the new a_0 and a_1 are driven to the respective registers in the state a subpart. The byte a_1, as already noted, is also the output of the algorithm.

2.4. Proposed 4-Bit Architecture of Enocoro-128v2 Stream Cipher

The proposed 4-bit architecture, for the Enocoro-128v2 stream cipher, is shown in Figure 3. The 4-bit architecture for the Enocoro-128 v2 stream cipher can be split in the same subparts as the 8-bit architecture. These subparts, however, differ greatly in their internal architecture. Again, the same initialization step is required before the first round of computations. The only difference is that in the 4-bit architecture each byte of data is fed into the buffer in 4-bit groups, beginning with the four least significant bits, and following the same procedure as in the 8-bit architecture, all the necessary bytes are present in buffer b after 48 clock cycles. The storage buffer is based on the same principles as the one in the 8-bit architecture. In order to achieve the necessary byte-wise right rotation of the buffer, each 4-bit group is rotated by 50 places (25 bytes) to the right at each clock cycle (each round requires 9 clock cycles).

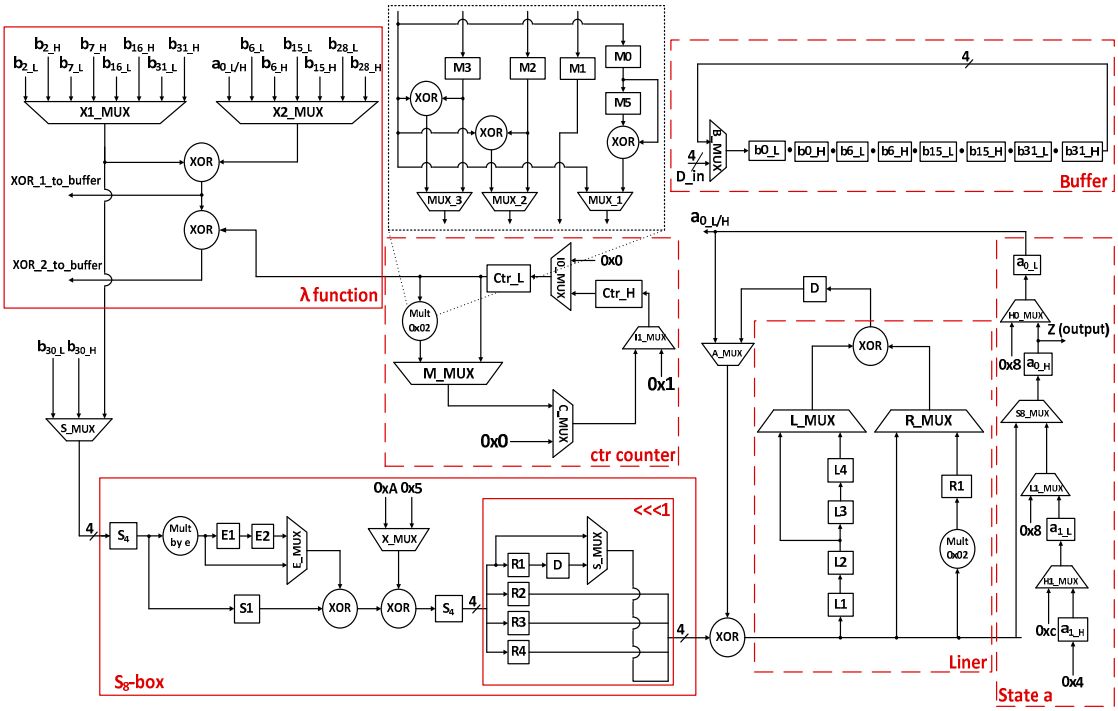

Figure 3. The proposed 4-bit architecture of Enocoro-128v2 stream cipher.

2.4.1. Function λ of the 4-Bit Architecture

The λ function in the 4-bit architecture works in the same way as the one in the 8-bit architecture. The only difference is that now the XORings are computed by 4-bit parts instead of complete bytes. Each XORing begins with the 4 LSBs of the specified byte. The ctr counter consists of two 4-bit registers, ctr_L and ctr_H, with initial values 0x0 and 0x1, respectively. One after the other, the values of those registers are driven to the multiplication subunit in order for the overall counter's value to be multiplied by 0x02. The result of this multiplication is stored in the corresponding registers in the final clock cycles of the current round, in order to the ctr counter's registers has the correct values at the

beginning of the next round. After the completion of 96 initialization rounds, the value 0×0 is stored in both the ctr counter's registers, which then remain unchanged.

The multiplication by 0x02 subunit has a latency of one clock cycle and its operation lasts for three clock cycles for every complete multiplication. In the first clock cycle the 4 LSBs of the byte that is to be multiplied enter the circuit and are stored in registers M0, M3, M2, and M1. In the second clock cycle the 4 MSBs of the byte also enter the circuit. The MSB of the four is separately XORed with the contents of M3 and M2. The results along with the MSB and the contents of register M1 are driven to the output and represent the four LSBs of the multiplication's result. In parallel, the MSB is stored in register M0 and the previous contents of M0 are moved to M5. The rest of the incoming bits are stored in registers M3, M2, and M1. In the third round, the contents of M0 and M5 are XORed. The result along with the contents of M3, M2, and M1 are driven to the output, as they represent the four MSBs of the multiplication's result.

2.4.2. Update Function ρ of the 4-Bit Architecture

The update function ρ of the 4-bit architecture consists of the same two parts as the one in the 8-bit architecture, the S_8-box part, and the Linear Transformation (Liner) part. For both these parts, novel internal 4-bit architectures were designed and implemented.

The S_8-box has a latency of two clock cycles and its operation, for each byte, lasts for four clocks cycles. In the first clock cycle, the 4 LSBs of the specified byte enter the S_8-box and are driven to the S_4-box subpart. The output is used in two distinct ways: it is stored in register S1 and is multiplied by the coefficient e = 0x04 (which is defined over the chosen finite field in GF(2^4)). The result of the multiplication is stored in register E1. In the second clock cycle, the remaining 4 MSBs of the specifying byte enter the S_8-box and are treated in the same way. The previous output that was stored in S1, along with the new multiplication result are driven to the first XOR gate. The result is driven to the second XOR gate along with coefficient 0×5. The gate's output is fed into the second S_4-box and the four output bits are stored in registers R1, R2, R3, and R4. Additionally, the contents of E1 are moved to E2. In the third clock cycle, the contents of S1 are driven to the first XOR gate along with the contents of E2. The gate's output is then driven to the second XOR gate along with the coefficient 0×A. The result is driven to the second S_4-box. The MSB of the output along with contents of R2, R3, and R4 are concatenated (as shown in Figure 3) and are driven to the output. The three reaming bits of the S_4-box output are stored in R2, R3, and R4 while the contents of R1 are moved to register D. In the fourth and final round, the contents of registers D, R2, R3, and R4 are concatenated and driven to the output. The output of the S_8-box is XORed with the appropriate four state bits (a_0_L, a_0_H, a_1_L, or a_1_H) and is fed to the Liner. This procedure continues for all the incoming bytes, in 4-bit parts, according to the algorithm's specifications.

The Liner has a latency of two clock cycles and its operation lasts for six clocks cycles. In the first clock cycle, the 4 LSBs of the first byte (Byte1 [3:0]) are fed into the Liner and are stored in register L1. In the second clock cycle, Byte1 [3:0] is moved to register L2 and the remaining 4 MSBs of the first byte (Byte1 [7:4]) are fed into the Liner and stored in L1. In the third clock cycle, the 4 LSBs of the second byte (Byte2 [3:0]) enter the Liner and are fed into the multiplication circuit (this multiplication by 0x02 circuit has already been described in the previous part of this paper). Additionally, through the right multiplexer, Byte2 [3:0] is driven to the XOR gate, while through the left multiplexer Byte1 [3:0] is also driven to the XOR gate. The result is then driven to the output. In the same clock cycle, Byte1 [3:0] is also moved to register L3 and Byte1 [7:4] is moved to L2. In the fourth clock cycle, the 4 MSBs of the second byte (Byte2 [7:4]) enter the Liner and are fed into the multiplication circuit. Again, through the right multiplexer, Byte2 [7:4] is driven to the XOR gate, while through the left multiplexer Byte1 [7:4] is also driven to the XOR gate. The result is driven to the output. Following the previous procedure, Byte1 [3:0] is moved to register L4 and Byte1 [7:4] is moved to L3. Additionally, the output of the multiplication circuit is stored in register R1. In the fifth clock cycle, the contents of L4 and R1 are XORed and the result is outputted. The Byte1 [7:4] is moved to L4 and the new output of the

multiplication circuit is stored in register R1. In the sixth and final clock cycle, the contents of L4 and R1 are XORed and the result is driven to the output. At the appropriate clock cycles, the Liner's output is XORed with the corresponding output of the S_8-box, according to the algorithm's specifications. The results which correspond to the new a_0_L, a_0_H, a_1_L, and a_1_H are driven to the respective registers that form the internal state a. The bits that correspond to a_1_L and a_1_H are also the output of the algorithm.

3. Results

The design suite that was selected for the synthesis and the implementation of the proposed 8-bit and 4-bit architectures of the Enocoro-128v2 stream cipher was Xilinx's Vivado 2019.2 through the use of the VERILOG hardware description language. The implementation was evaluated (routed and placed) in the Basys 3 Artix-7 (XC7A35T) FPGA Board. The implementation results of both the proposed architectures are shown in Table 2.

Table 2. Implementation Results.

FPGA Device	XC7A35T (BASYS3 ARTIX-7)	
Architecture's Datapath	8-bit	4-bit
Registers (FFs)	330	343
Look-up tables (LUTs)	254	249
Frequency (MHz)	189	204
Throughput (Mbps)	302	181
# Slices	78	83
Throughput /#Slices	3.8	2.2
Round's latency	5	9
On-chip dynamic Power (mW)	41	40

The 8-bit implementation utilizes 254 LUTs (1.22% of the available LUTs) and 330 FFs (0.79% of the available FFs) in a total of 78 slices. Each round of the algorithm requires 5 clock cycles for its completion, while the maximum reached frequency is 189 MHz, leading to a throughput equal to 302 Mbps. The achieved throughput per number of slices for the 8-bit implementation is 3.8. On the other hand, the 4-bit architecture achieves a clock frequency of 204 MHz, a throughput equal to 181 Mbps with a latency per round equal to 9 clock cycles, while it utilizes only 343 FFs (0.82% of the available FFs) and 249 LUTs (1.19% of the available LUTs) in a total of 83 slices. The throughput per number of slices for the 4-bit implementation is 2.2. In addition, we estimated the on-chip power using the Vivado Report Power feature. For accurate and reliable estimation, the Switching Activity Interchange format (SAIF) file, which is generated from Post Implementation Timing Simulation, was used. The SAIF file mainly contains the signals' toggle counts (number of changes). In order to achieve a good toggle coverage in the design's internal signals (which leads to more accurate estimation), a very long simulation time was necessary e.g., 100.000 ns (with the clock period being 5 ns). Additionally, the inputs for the implementations were generated by a random number generator. We estimated that the 8-bit implementation has a dynamic power dissipation of 41 mW while the dynamic power dissipation of the 4-bit architecture is 40 mW. As it is well known, the static power consumption mainly depends on the total resources of the FPGA rather than the number of the resources that are utilized by an implementation. So, the static power dissipation of the device is estimated to be equal to 70 mw for both architectures.

4. Discussion

In Table 3, some comparison metrics regarding area and performance with previously published compact FPGA implementations for lightweight stream ciphers are given. These implementations are the closest ones that can be used for comparisons with the proposed implementations, due to the fact that implementations for the Enocoro-128v2 stream cipher are very few. The comparisons are focused

on Frequency (Freq.), the Throughput (Mbps), the Area (#Slices) and, finally, the Throughput per area (T/#Slices) that measures the hardware resource cost associated with the implementation's throughput (the higher value is the better). Due to the fact that the stream ciphers that were used for comparisons were implemented in different FPGAs, and this can lead to a potentially different maximum frequency and hardware resource utilization, we believe that the best metric for comparison is the Throughput per area (T/#Slices) because it can mitigate the previously mentioned potential differences. We have denoted the proposed 8-bit architecture as Enocoro8 and the proposed 4-bit architecture as Enocoro4 for simplicity.

Table 3. Comparisons of the proposed Enocoro-128v2 architectures with implementations of other cryptographic algorithms.

Cipher	Freq. (MHz)	Throughput (Mbps)	# Slices	T/#Slices	FPGA Board
DECIM v2 [13]	185	46	80	0.58	Spartan-3
DECIM-128 [13]	174	43	89	0.49	Spartan-3
E0 [14]	187	187	140	1.33	Spartan-3
Grain v1 [13]	196	196	44	4.45	Spartan-3
Grain v1 [14]	177	177	318	0.55	Spartan-3
Grain v1 [15]	193	193	122	1.58	Spartan-3
Grain v1 [16]	-	105	48	2.19	Spartan-II
Grain-128 [13]	196	196	50	3.92	Spartan-3
Grain-128 [17]	181	181	48	3.77	Virtex-II
MICKEY 2.0 [13]	233	233	115	2.03	Spartan-3
MICKEY 2.0 [14]	250	250	98	2.55	Spartan-3
MICKEY-128 2.0 [13]	223	223	176	1.27	Spartan-3
MICKEY-128 2.0 [15]	156	156	261	0.60	Spartan-3
MICKEY-128 2.0 [17]	200	200	190	1.05	Virtex-II
MICKEY-128 2.0 [18]	170	170	167	1.02	Virtex
Moustique [13]	225	225	278	0.81	Spartan-3
Moustique [19]	-	369	252	1.46	Virtex-II
Mosquito [16]	-	137	298	0.46	Spartan-II
Trivium [13]	240	240	50	4.80	Spartan-3
Trivium [14]	326	326	149	2.18	Spartan-3
Trivium [15]	201	201	188	1.07	Spartan-3
Trivium [16]	-	102	40	2.55	Spartan-II
Trivium [17]	207	207	41	5.05	Virtex-II
Enocoro-128v2 [20]	118	-	292	0.40	Spartan-3
Enocoro-128v2 [20]	149	-	442	0.33	Spartan-3
Enocoro8	189	302	78	3.87	ARTIX 7
Enocoro4	204	181	83	2.18	ARTIX 7

As already discussed, in [13] an implementation of DECIM v2 and an implementation of DECIM-128 are proposed. Both our enocoro implementations achieve higher maximum frequencies, higher throughputs and have higher T/#Slices factors than both the DECIM ones. Our 8-bit Enocoro implementation utilizes less slices than both the DECIM ones while our 4-bit Enocoro implementation utilizes 6 slices less than the DECIM-128 implementation and only 3 more than the one of the DECIM v2. Compared to the implementation of E0 that is presented in [14], our 8-bit Enocoro implementation achieves a higher frequency, a higher throughput, it utilizes less slices and has a better T/#Slices factor. Our 4-bit Enocoro implementation achieves a higher frequency, it utilizes less slices and has a better T/#Slices factor, but it reaches a slightly lower maximum frequency (6 MHz less).

Regarding the four implementations of Grain v1 that are presented in [13–16], our 8-bit Enocoro implementation achieves a higher operating frequency compared to [14] but lower than [13,15]. The throughput of our implementation is higher than all Grain v1 implementations. Regarding the slice usage, our 8-bit implementation utilizes fewer slices than [14] and [15] but more than [13,16]. The T/#Slices factor of our implementation is better than [14–16] but worse than [13]. Our 4-bit Enocoro implementation achieves a higher operating frequency compared to all Grain v1 implementations. The throughput of our implementation is higher than [14,16] but lower than [13,15]. As for the slice

usage, our 4-bit implementation utilizes fewer slices than [14,15] but more than [13,16]. The T/#Slices factor of our implementation is better than [14,15] but worse than [13,16].

As for the two implementations of Grain-128 that are presented in [13,17], our 8-bit Enocoro implementation achieves a higher frequency than the one in [17] but lower than [13], while it achieves higher throughput than both the Grain-128 implementations. However, it utilizes more slices than both the Grain-128 ones. This leads to a better T/#Slices factor than [17] but worse than [13]. On the other hand, our 4-bit implementation has a frequency higher than both [13,17], a throughput equal to [17] but lower than [13], it utilizes more slices than both the Grain-128 implementations and its T/#Slices factor is lower than both [13,17].

In comparison to the two MICKEY 2.0 implementations in [13,14], our 8-bit Enocoro implementation reaches a lower frequency than both of them, but it has a higher throughput than both. Additionally, it utilizes fewer slices and has a better T/#Slices factor than both the MICKEY 2.0 implementations. Our 4-bit Enocoro implementation reaches a lower frequency and a lower throughput than both of the MICKEY 2.0 implementations. However, at the same time, it utilizes fewer slices than both of them and it has a T/#Slices factor that is higher than [13] but lower than [14].

Moreover, compared to the four implementations of MICKEY-128 2.0 that are proposed in [13,15,17,18], our 8-bit Enocoro implementation achieves a higher operating frequency compared to [15,18] but lower than [13,17]. The throughput of our implementation is higher than all MICKEY-128 2.0 implementations. Additionally, our 8-bit Enocoro implementation utilizes fewer slices and has a better T/#Slices factor than all MICKEY-128 2.0 implementations. On the other hand, our 4-bit implementation has a higher operating frequency compared to [15,17,18] but lower than [13]. Regarding the throughput, our implementation achieves a higher throughput compared to [15,18] but lower than [13,17]. Again, our 4-bit Enocoro implementation utilizes fewer slices and has a better T/#Slices factor than all MICKEY-128 2.0 implementations.

As already presented, two implementations of Moustique are proposed in [13,19]. The Moustique in [13] achieves a frequency higher than both our Enocoro implementations, it has lower throughput than our 8-bit implementation but higher than our 4-bit one. However, both our implementations utilize significantly less slices and have much better T/#Slices factors. The Moustique in [19] achieves a throughput higher than both our Enocoro implementations but, at the same time, it utilizes significantly more slices and has a much worst T/#Slices factor than both our implementations. Against the Mosquito implementation in [16], both our enocoro implementations perform better on all the corresponding metrics.

Regarding the five implementations of Trivium that are presented in [13–17], our 8-bit Enocoro implementation achieves a frequency that is lower than all Trivium implementations. However, it achieves a maximum throughput that is higher than [13,15–17] but lower than [14]. Regarding the slice usage, our 8-bit implementation utilizes fewer slices than [14,15] but more than [13,16,17]. The T/#Slices factor of our implementation is better than [14–16] but worse than [13,17]. Our 4-bit Enocoro implementation achieves a higher operating frequency compared to [15] but lower compared to the rest of the Trivium implementations. It achieves a maximum throughput that is higher than [16] but lower than the rest. Additionally, it utilizes fewer slices than [14,15] but more than [13,16,17]. The T/#Slices factor of our 4-bit Enocoro implementation is the same as [14], better than [15] but worse than [13,16,17].

Finally, compared to the two implementations of the Enocoro-128v2 that are presented in [20], both our Enocoro-128v2 implementations achieve higher max frequencies, utilize less slices, and have higher T/#Slices factors.

From the previous comparisons of our 8-bit and 4-bit Enocoro implementations with a large variety of stream ciphers, it can be seen that our implementations compared very well against them. Even against implementations that have a different design philosophy like the compact Grain and Trivium, our implementations exhibit very good results. The lightweight stream ciphers are designed for area constraint embedded devices. That is why they generally consume fewer hardware resources

and achieve better performance compared to conventional block ciphers such as the AES, the Triple Data Encryption Algorithm (3DES), etc. In addition, the constrained embedded devices have limited information processing resources in their CPU or in their memory and more importantly they have restricted low power requirements.

Therefore, efficiency in hardware means a balance between the previous mentioned factors. We strongly believe that the proposed architectures achieve a very good level of efficiency in terms of the previous factors because they does not utilize any memory, they require a low number of hardware resources and, therefore, do not consume a lot of power while simultaneously achieving a very good level of time efficiency.

5. Conclusions and Future Works

Two very compact architectures of the lightweight stream cipher Enocoro-128v2, along with their implementations, are proposed in this paper. The Enocoro-128v2 stream cipher is part of the ISO/IEC 29192-3 standard. The 8-bit architecture achieves a throughput equal to 302 Mbps @ 189 MHz while the 4-bit architecture achieves a throughput equal to 181 Mbps @ 204 MHz. Both architectures utilize a very low number of hardware resources that leads them to also have very low dynamic power consumption, specifically, up to 41 mW for the 8-bit one and 40 mW for the 4-bit one. Comparisons in terms of operating frequency, area, and most importantly in throughput per area, with the most well-known lightweight stream ciphers, prove that the proposed architectures are a very good candidate for the security aspect of area embedded devices.

Future works will aim to improve the proposed architectures in the architectural design level and also explore different options of implementation. Regarding the part of the implementation, our goal is to implement the proposed architectures in silicon and in miniature FPGAs in order to further reduce the power consumption to μW and enable the efficient integration to even more compact IoT nodes and constrained embedded systems in general. At architectural level, we will aim to further explore FPGA low power techniques (such as glitch reduction or signal gating) in order to, again, further reduce the power consumption. In this regard, a very important technique for power reduction is the Guarded Evaluation [21] because it can stop the input switching activity from propagating when the outputs are not used. This feature is common in feedback logic designs like our proposed architectures. In addition, because embedded applications now tend to use parallel computing architectures of algorithms like Enocoro 128v2 that already have a high degree of parallelism, they are a primary option for the devices' security module. Finally, additional features against side channel analysis attacks will be designed and included in the proposed architectures, such as countermeasures based on secure logic styles, hiding countermeasures, and masking countermeasures.

Author Contributions: Conceptualization, L.P. and P.K.; methodology, L.P. and P.K.; software, L.P.; validation, L.P. and P.K.; formal analysis, P.K.; investigation, L.P. and P.K.; resources, P.K.; data curation, L.P. and P.K.; writing—original draft preparation, L.P.; writing—review and editing, L.P. and P.K.; visualization, L.P. and P.K.; supervision, P.K.; project administration, P.K.; funding acquisition, P.K. All authors have read and agreed to the published version of the manuscript.

Funding: We acknowledge support of this work by the project "I3T—Innovative Application of Industrial Internet of Things (IIoT) in Smart Environments" (MIS 5002434) which is implemented under the "Action for the Strategic Development on the Research and Technological Sector", funded by the Operational Programme "Competitiveness, Entrepreneurship and Innovation" (NSRF 2014–2020) and co-financed by Greece and the European Union (European Regional Development Fund).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alguliyev, R.; Imamverdiyev, Y.; Sukhostat, L. Cyber-physical systems and their security issues. *Comput. Ind.* **2018**, *100*, 212–223. [CrossRef]
2. Walker-Roberts, S.; Hammoudeh, M.; Aldabbas, O.; Aydin, M.; Dehghantanha, A. Threats on the horizon: Understanding security threats in the era of cyber-physical systems. *J. Supercomput.* **2020**, *76*, 2643–2664. [CrossRef]
3. Nazarenko, A.A.; Safdar, G.A. Survey on security and privacy issues in cyber physical systems. *AIMS Electron. Electr. Eng.* **2019**, *3*, 111–143. [CrossRef]
4. Eisenbarth, T.; Kumar, S.S.; Paar, C.; Poschmann, A.; Uhsadel, L. A Survey of Lightweight-Cryptography Implementations. *IEEE Des. Test. Comput.* **2007**, *24*, 522–533. [CrossRef]
5. Abed, S.; Jaffal, R.; Mohd, B.J.; Alshayeji, M. FPGA Modeling and Optimization of a SIMON Lightweight Block Cipher. *Sensors* **2019**, *19*, 913. [CrossRef] [PubMed]
6. Medien, Z.; Machhout, M.; Bouallegue, B.; Khriji, L.; Baganne, A.; Tourki, R. Design and Hardware Implementation of QoSS-AES Processor for Multimedia applications. *Trans. Data Priv.* **2010**, *3*, 43–64. [CrossRef]
7. Buchanan, W.J.; Li, S.; Asif, R. Lightweight cryptography methods. *J. Cyber Secur. Technol.* **2017**, *1*, 187–201. [CrossRef]
8. Watanabe, D.; Okamoto, K.; Kaneko, T. A Hardware-Oriented Light Weight Pseudo-Random Number Generator Enocoro-128v2. In Proceedings of the 2010 Symposium on Cryptography and Information Security, SCIS 2010, Okayama, Japan, 8–12 December 2010. 3D1-3.
9. Watanabe, D.; Owada, T.; Okamoto, K.; Igarashi, Y.; Kaneko, T. Update on Enocoro Stream Cipher. In Proceedings of the International Symposium on Information Theory and Its Applications (ISITA), Taichung, Taiwan, 17–20 October 2010; pp. 778–783. [CrossRef]
10. ISO/IEC 29192-3:2012. Information Technology—Security Techniques—Lightweight Cryptography—Part 3: Stream Ciphers. 2012. Available online: https://www.iso.org/standard/56426.html (accessed on 20 August 2020).
11. International Electrotechnical Commission (IEC). News Release 2012: Number 19. Available online: https://www.iec.ch/newslog/2012/nr1912.htm (accessed on 20 August 2020).
12. Daemen, J.; Clapp, C. Fast Hashing and Stream Encryption with Panama. In *Fast Software Encryption (FSE'98)*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 1998; Volume 1372, pp. 60–74. [CrossRef]
13. Hwang, D.; Chaney, M.; Karanam, S.; Ton, N.; Gaj, K. Comparison of FPGA-targeted Hardware Implementations of eSTREAM Stream Cipher Candidates. State Art Stream Ciphers Workshop SASC. 2008, pp. 151–162. Available online: http://www.ecrypt.eu.org/stream/ (accessed on 20 August 2020).
14. Kitsos, P.; Sklavos, N.; Provelengios, G.; Skodras, A.N. FPGA-based performance analysis of stream ciphers ZUC, Snow3g, Grain V1, Mickey V2, Trivium and E0. *Microprocess. Microsyst.* **2013**, *37*, 235–245. [CrossRef]
15. Gaj, K.; Southern, G.; Bachimanchi, R. Comparison of Hardware Performance of Selected Phase II eSTREAM Candidates. State of the Art of Stream Ciphers Workshop (SASC 2007). eSTREAM. ECRYPT Stream Cipher Project. Report 2007/026. 2007. Available online: https://www.ecrypt.eu.org/stream/papersdir/2007/026.pdf (accessed on 20 August 2020).
16. Good, T.; Chelton, W.; Benaissa, M. Review of Stream Cipher Candidates from a Low Resource Hardware Perspective. eSTREAM. ECRYPT Stream Cipher Project. Report 2006/016. 2006. Available online: https://www.ecrypt.eu.org/stream/papersdir/2006/016.pdf (accessed on 20 August 2020).
17. Bulens, P.; Kalach, K.; Standaert, F.X.; Quisquater, J.J. FPGA Implementations of eSTREAM Phase-2 Focus Candidates with Hardware Profile. State of the Art of Stream Ciphers Workshop (SASC 2007). eSTREAM. ECRYPT Stream Cipher Project. Report 2007/026. 2007. Available online: https://www.ecrypt.eu.org/stream/papersdir/2007/024.pdf (accessed on 20 August 2020).
18. Kitsos, P. On the Hardware Implementation of the MICKEY-128 Stream Cipher. eSTREAM. ECRYPT Stream Cipher Project. Report 2006/059. 2006. Available online: https://www.ecrypt.eu.org/stream/papersdir/2006/059.pdf (accessed on 20 August 2020).
19. Daemen, J.; Kitsos, P. The self-synchronizing stream cipher MOUSTIQUE. In *New Stream Cipher Designs—The eSTREAM Finalists*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 210–223. Available online: http://www.ecrypt.eu.org/stream/mosquitop3.html (accessed on 20 August 2020).

20. Manifavas, C.; Hatzivasilis, G.; Fysarakis, K.; Papaefstathiou, Y. A survey of lightweight stream ciphers for embedded systems. *Secur. Comm. Netw.* **2016**, *9*, 1226–1246. [CrossRef]
21. Ravishankar, C.; Anderson, J.; Kennings, A. FPGA power reduction by Guarded Evaluation considering logic architecture. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2012**, *31*, 1305–1318. [CrossRef]