SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Implementing ChaCha20: analysis on performance, resource utilization and side-channel protection

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Vittorio Dani, 995572

Advisor:
Prof. Alessandro Barenghi
Academic year:
2023-2024

Abstract: ChaCha20 is a symmetric stream cipher which involves fast and cheap operations compared to AES. In this work we propose an hardware implementation of the cipher based on fully-combinatorial quarter-round units, then we explore three different architectural optimizations, i.e., smaller component size, loop unrolling and pipelining. We explore a protected design of the cipher from side-channel attacks exploiting the threshold implementation tecnique, proposing masking schemes for its arithmetic components, then we compare it with an existing gate-level masking countermeasure. We report the figures of merit of these implementations targeting an Artix-7 FPGA, reaching 5.62 Gb/s with a pipelined design and 1.79 Mb/s/slice with a combinatorial design. We evaluate the security of protected designs by means of statistical power analysis.

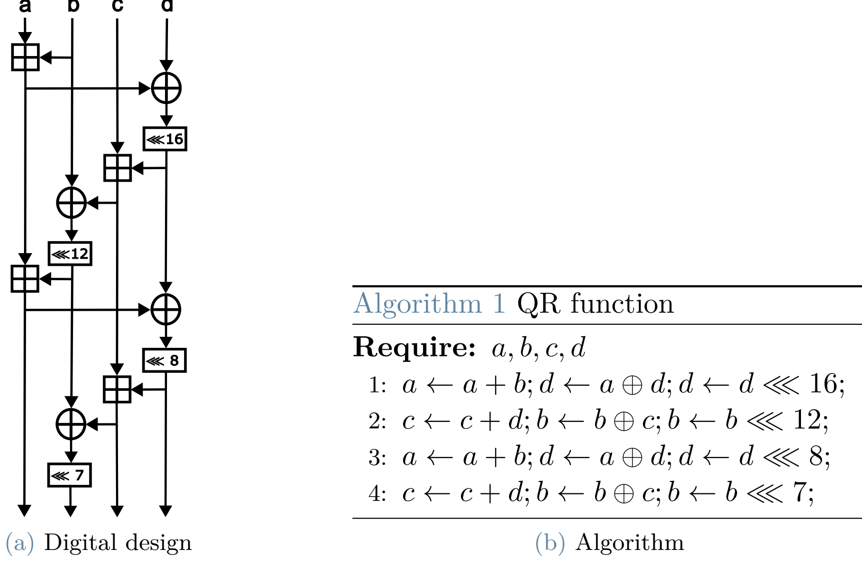
Key-words: ChaCha20, cipher, side-channel, masking, FPGA

1. Introduction

As for the implementations of ciphers, the ability to implement fast and resource-efficient hardware cryptographic devices is critical, particularly in systems where processing power, memory and hardware resources are often severely limited, such as embedded systems, which are frequently used in scenarios where information security is critical. Another important feature of ciphers is their resistance to side-channel attacks. Side-channel analysis is a type of cryptanalytic attacks that exploit the physical properties of a cryptosystem to derive some leaks about its secrets, for example by analysing the power consumption or the electromagnetic radiations [26]. A side-channel-resistant cipher encrypts data without physically leaking any sensitive information, that is, physical measurements during an encryption operation are unrelated to the secret key.
ChaCha20 is a stream cipher published by Daniel J. Bernstein in 2008, introduced in [3]. In this cipher, the keystream is generated using a block function with a counter mode of operation, where successive values of a counter are encrypted, and the resulting keystream is combined by XOR operation with the plaintext. ChaCha20 is a low-cost and high-speed cipher, due to the fact that it is an ARX algorithm; in other words, its round function involves as operations only modular addition, rotation and XOR, which are fast and cheap in hardware and software. This is in contrast with the current standard for symmetric cryptography, the Advanced Encryption Standard (AES), which in the absence of dedicated hardware has relatively poor performance [24]. These features suggest ChaCha20 as a desirable alternative to AES as a symmetric encryption system.
The goal of this thesis is to explore the ChaCha20 cipher in terms of performance, resources utilization and side-channel protection. To do this, in Chapter 1 we begin by describing in detail the mechanism of the cipher,

constant	constant	constant	constant
key	key	key	key
key	key	key	key
counter	nonce	nonce	nonce

Figure 1: ChaCha20 intial matrix, each cell represents a 32-bit value



(a) Digital design (b) Algorithm

Figure 2: QR function, inputs a, b, c, d are 32-bit words

then we review previous works about ChaCha20 implementations and we introduce the problem of side-channel attacks. In Chapter 2 we describe our basic design for the cipher and we propose three architectural optimizations which achieve different trade-offs between performance and resources utilization: sequential design with reduced components size, unrolled and pipelined design; then we propose a side-channel resistant implementation of the cipher that exploits a masking technique called threshold implementation, demonstrate mathematically its security, and illustrate a second low-cost implementation as a comparison term. Although masking schemes, including threshold implementations, for various types of adders already exist, we did not find in the literature our proposed threshold implementation of the ripple-carry adder. Finally, in Chapter 3 we collect hardware synthesis data of each cipher implementation; in particular, we report timing data, resources utilization, performance, critical paths and analyze the power consumption of the protected implementation, comparing all results each other and with existing works, both about ChaCha20 and AES implementations.

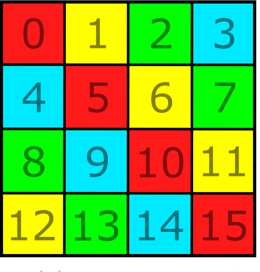
2. State of the art

In this section we recall the structure of the ChaCha20 cipher, we review existing works about ChaCha20 implementations and we describe side-channel attacks and countermeasures.

2.1. Structure of the ChaCha20 symmetric cipher

We now summarize the working mechanism of the ChaCha20 cipher, as shown in [3].
ChaCha20 is an algorithm that acts on a 4×4 matrix of 32-bit values known as *state*. Initially the first four words of the state come from blocks of four characters that belong to an ASCII string, the next eight words are a 256-bit key, ninth and tenth words are a 64-bit counter, the last two words are a 64-bit nonce; from this moment we refer to this configuration as *initial state*. In our work we implement the cipher variant described in [24], where the counter is a 32-bit value and the nonce is a 96-bit value; this change makes the cipher compatible with the nonce formation rule recommended in [23]. The resulting initial state is shown in Figure 1.
The primitive operation of ChaCha20 is the Quarter-Round (QR). It receives as input four 32-bit state words a, b, c, d, and transforms them by applying four times in sequence the operations of addition, XOR and rotation, as described in Figure 2.

(a) Column round



(b) Diagonal round

Figure 3: Column and diagonal rounds, each cell represents a 32-bit word of the state, cells of the same color are subject to the same QR operation

---

Algorithm 2 ChaCha20 block function

Require: $in[16]$
Ensure: $out[16]$
1: $x \leftarrow in$
2: **for** $i = 0$ to 15 **do**
3: $\quad x[i] \leftarrow in[i]$
4: **end for**
5: **for** $i = 1$ to 10 **do**
6: $\quad \mathrm{QR}(x[0], x[4], x[8], x[12])$ ▷ Column round
7: $\quad \mathrm{QR}(x[1], x[5], x[9], x[13])$
8: $\quad \mathrm{QR}(x[2], x[6], x[10], x[14])$
9: $\quad \mathrm{QR}(x[3], x[7], x[11], x[15])$
10: $\quad \mathrm{QR}(x[0], x[5], x[10], x[15])$ ▷ Diagonal round
11: $\quad \mathrm{QR}(x[1], x[6], x[11], x[12])$
12: $\quad \mathrm{QR}(x[2], x[7], x[8], x[13])$
13: $\quad \mathrm{QR}(x[3], x[4], x[9], x[14])$
14: **end for**
15: **for** $i = 0$ to 15 **do**
16: $\quad out[i] \leftarrow x[i] + in[i]$
17: **end for**

ChaCha20 block function transforms the initial state through 20 rounds, which consist of four quarter-rounds, alternating between *column rounds* and *diagonal rounds*. In Figure 3 a column round and diagonal round are shown; each cell represents a 32-bit word of the state, cells of the same color are subject to the same QR operation. An important aspect is that the four QR operations within each round all operate on different state words, consequently they can be parallelized. The pseudocode of ChaCha20 block function is shown in Algorithm 2.
Finally the block function adds the result to the initial state to obtain a 16-word output block.
ChaCha20 cipher takes as inputs a 256-bit key, a 32-bit initial counter, a 96-bit nonce and an arbitrary-length plaintext. Then it successively calls the ChaCha20 block function, with the same key and nonce, and with an increasing block counter; concatenating the resulting blocks forms a keystream. The ChaCha20 function then performs an XOR of this keystream with the plaintext, obtaining a ciphertext which has the same length; if there is extra keystream from the last block, it is discarded.
Decryption is done in the same way, obviously using the ciphertext instead of the plaintext. Note that the nonce is an arbitrary number which should not be repeated for the same key.

2.2. Related works

There are already some works that have presented implementations of the ChaCha20 cipher. In [14], Henzen et al. present the first hardware implementations of ChaCha. As opposed to later proposed works which are FPGA-based, these are VLSI implementations synthesized using a 0.18 ţm CMOS technology. Several variants are explored: implementations with 1 QR, 4 parallel QRs and 8 QRs which compute combinatorially two rounds in one clock cycle; also, QR modules fully implemented and with QR modules that integrate only a single ARX

stage are explored. In that work the peak performance is obtained with the 8-QR variant, which reaches a throughput of 6.8 Gb/s. In [25], Pfau et al. compare three different architectures for the cipher: pipeline-based, block memory and register implementations. The pipeline-based implementation provides a set of round blocks composed of four parallel pipelined quarter-round cells, in turn each one composed of four pipelined ARX stages; the round blocks are chained to form a long pipeline. In the block memory implementation one pipelined quarter-round cell is used, the four input and output words of a round are respectively read and saved in parallel inside four block RAMs at precise memory addresses. Finally, in the register implementation the state is mapped into registers, four ARX stages of as many independent QR operations are computed in parallel, computing one round in four clock cycles, and each intermediate result is saved into registers. A Verilog implementation of ChaCha is proposed by Strömbergson et al. in [32], in which the top modules manages four parallel QR modules, computing one round per clock cycle. Xiphera [34] offers a commercial implementation of ChaCha20-Poly1305, a combination of ChaCha20 cipher and Poly1305 message authentication code; since it is a closed-source IP-core, implementation details are not available. In [2], At et al. describe an architecture based on a control unit, a register file and an ALU which is deeply pipelined in order to reduce the critical path; in order to make the most of the pipeline, independent computations are accurately overlapped. This implementation is able to reach 266 Mb/s by using as few as 49 slices and 2 BRAMs on Virtex-6. In [7], Cuppens et al. describe an implementation in which an input and an output block exchange data with the memory, and, between these blocks, the ChaCha20 processing block calculates one ciphertext in 83 clock cycles using four quarter-round units in parallel, which togheter compute one round in 5 clock cycles. In [29], Serrano et al. presents an architecture decomposed into a finite state machine, which manages the encryption process, a block function with four combinatorial quarter-round units, each one composed of four ARX cells, and a cryptographic function, which uses the keystreams to generate the ciphertext.
Compared with these works, ours shares some similarities and at the same time is different. From the technological point of view, compared to [14], we consider FPGA as target instead of ASIC, and, as opposed to [25], we don't make use of block RAMs, instead we always save data inside registers. From the digital design perspective, with respect to [14] we go beyond the double-round per clock cycle described in that work by exploring with our unrolled designs up to a fully-combinatorial 20-round computation. We also achieve a different trade-off between area occupation and maximum clock frequency by executing one arithmetic operation per clock cycle instead of an ARX stage. The works [32] and [29] strictly focus on a 4-QR combinatorial design, instead we explore different QR implementations, i.e., QR modules with reduced components size, and different numbers of QR modules. The work [7] explores a sequential QR design as we do, but we also implement components of different sizes in order to evaluate the impact on resources occupation. Finally, with respect to the pipelined designs in [2] and [25] we build a different, less deep pipeline structure, in which each stage executes one complete round, with a number of stages from 2 to 21.

2.3. Side-channel attacks

In classical cryptanalysis a cipher is modeled as a mathematical object which receives an input value, transforms it according to a secret parameter, and returns it as output; information about the secret parameter can be obtained by exploiting the mathematical properties of that object. Conversely, it is possible to exploit the characteristics of the physical implementation of a cipher to obtain the secret parameter; these type of attacks are called physical attacks. In particular, the side-channel attacks are a class of physical attacks in which information is derived by observing physical features of the device which are linked to a secret parameter [31]. Mainly, side-channel attacks are classified in two ways:

- Active vs. passive: active attacks cause malfunctions to the device, while passive attacks merely observe the behavior of the device.
- Invasive vs. non-invasive: invasive attacks depackage the chip to gain access to internal components, while non-invasive attacks do not physically alter the chip.

In a cryptographic device there are different physical phenomena related to its computation that can be measured and exploited to derive sensitive information. Some of the main ones include power consumption [18], electromagnetic radiations [11], temperature [17] and computation time [19].
In this paper we will focus on power consumption and on electromagnetic radiations. In CMOS devices one important dissipation source is the dynamic power consumption, resulting from the charge and discharge of the load capacitance; consequently, this power consumption is directly linked to the internal data. In addition, since, currents mainly flow when there is a change in the logic state, according to the Biot-Savart law they result in compromising emanations, which as a consequence reveal the data inside the device. Two classical attacks are Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [18]: in SPA, power consumption is analyzed to derive the performed operations, while, in DPA, data dependencies in power consumptions are exploited.
Various types of countermeasures to side-channel attacks have been proposed. One typical solution is generating

noise to disturb the side channel. The leakage can be made inaccessible by physical schields. Dynamic logic styles alternative to CMOS have been proposed to break the link between power consumption and data. At the algorithmic level, time randomization can make timing attacks difficult, as well as any attack that requires precise synchronization. With data encryption side channels only reveal harmless information about ciphertexts. It's possibile to make the leakage constant, a practise known as hiding. An interesting tecnique is masking, which principle, as shown in [13], is to split every intermediate variable x into $d+1$ shares, where d is a security parameter named masking order; shares can be later recombined through addition in the original value. Equation 1 shows a decomposition of variable x into three shares:

$$x = x_0 + x_1 + x_2 \tag{1}$$

In the same way, any function F is split up into a number of component functions in such a way that the sum over the component functions gives the same result as the original function. Equation 2 shows a decomposition of the two-variable function F into three component functions:

$$F(x, y) = F_0 + F_1 + F_2 \tag{2}$$

In this work we exploit a particular masking technique named *threshold implementation* (TI). Threshold implementations focus on the properties the component functions have to fulfill in order to guarantee data independence [27]. These properties are:

- Correctness: the sum over the component functions must give the same result as the original function.
- Non-completeness: each component function is independent of at least one share per variable.
- Uniformity: all share inputs and outputs of component functions must be uniformly distributed regardless of which unshared values they represent; this can be achieved by using random bits, which are denoted by letter z.

These properties can be violated by the presence of glitches, which are created by different arrival times of signals. As a consequence, every component function output must be saved into a register before using it in the next function.

Another masking tecnique exploited in this work is the protection proposed in [20], which, compared to other common masking schemes, while being less secure, is expected to have a significantly lower cost. From this moment we refer to it as *low cost implementation* (LC). As opposed to the threshold implementation, in which masking acts at function level, in the low cost implementation masking is done at gate level; in other words, instead of computing masked functions, we build secured logic gates, on the basis of which we compute the normal functions. Here is the scheme of the protected AND gate, where $z = xy$ and each bit is decomposed into two shares:

$$\begin{cases} z_0 = (x_0 \wedge y_0) \oplus (x_0 \vee \neg y_1) \\ z_1 = (x_1 \wedge y_0) \oplus (x_1 \vee \neg y_1) \end{cases} \tag{3}$$

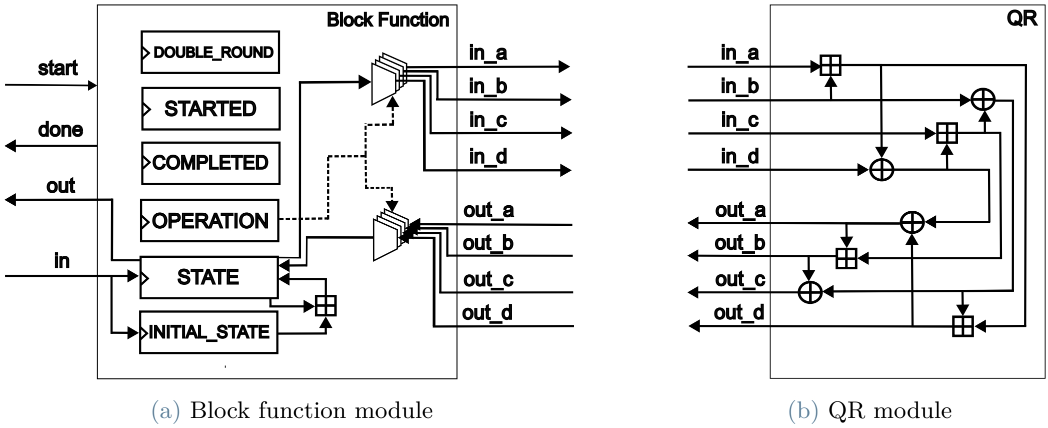
To counteract the effect of glitches, we take control over the order of the input signals in such a way that there is no information leakage. In the proposed implementation of the AND gate, both z_0 and z_1 depend on y_0 and y_1, therefore, as long as either y_0 or y_1 arrive last, we observe no leakage. As a consequence, by delaying y_1 with a flip-flop we can achieve a temporary non-completeness property for both output bits.

There exist several methodologies for evaluating the side-channel vulnerability in a cryptographic device. Test Vector Leakage Assessment (TVLA) is a widely used methodology for evaluating the side-channel attack resistance [35]. The core idea is that, in a protected design, both encrypting the same plaintext several times and encrypting several random plaintexts generate indistinguishable behaviors. Accordingly, traces of the physical phenomenon under observation during encryptions are acquired, distinguishing fixed input and random input traces into two sets, and are analyzed statistically by exploiting Welch's t-test of the means between two populations, validating the hypothesis that the two populations of traces have the same mean. In this case the test has a dual setting to the canonical test of statistics: the null hypothesis is that the means of the two populations are different, and it is discarded only if the data differ widely from that assumption. Given two trace sets A and B, where a_{ij} and b_{ij} are the i-th sample of the j-th trace of set A and B, respectively, the set of i-th samples A_i and B_i are interpreted as populations generated by sequences of random variables with distributions $f_{A_i}(x)$ and $f_{B_i}(x)$. The t-statistic on the i-th samples is computed as follows:

$$t_i = \frac{(\overline{T_{A_i}} - \overline{T_{B_i}})}{\sqrt{(S_{A_i}^2 / N_{A_i}) + (S_{B_i}^2 / N_{B_i})}} \tag{4}$$

where $\overline{T_X}$, S_X, N_X are the sample means, sample variances, and cardinality of samples set X, respectively.

For each point in time, if the value of t-statistic is beyond a threshold determined by the test significance level, the point is marked as leakage point, otherwise the point is marked as a nonleakage point.



(a) Block function module

(b) QR module

Figure 4: Block function and QR modules in the basic design

3. Cipher implementations

Our goal is to explore different implementations of the cipher in order to then compare relative figures of merit. Versions are made that integrate either a single QR module or four QR modules working in parallel; further distinctions are related to the size of components, i.e., adders and xor banks, which can be 32-bit, 16-bit or 8-bit, and the type of logic in the QR module, which can be sequential or combinatorial. Unrolled versions of the cipher are then explored, with factors of 2, 4, 5, 10 and 20, as well as pipelined versions, with 2, 4, 5, 10, 20 and 21 stages. Another area of exploration is protection from side-channel attacks; two different countermeasures are employed: a threshold implementation at the algorithm level and a low-cost implementation at the gate level. In this section we begin by describing our fundamental design, then list the considered architectural optimizations, and finally propose countermeasures for side-channel analysis.

3.1. Fundamental design

Our basic design includes, in hierarchical order from top to bottom: a cipher module, a block function module, and at least one quarter-round module. The block function module uses the quarter-round module to calculate the final state, while the cipher module saves data about key, nonce and counter, gets the state from the block function, and then encrypts the plaintext.
The QR module receives as input four 32-bit words, performs an entire quarter-round computation in a single clock cycle and returns the transformed four words as output. Each QR module integrates 4 32-bit adders and 4 32-bit xor banks connected to each other. The internal logic is simple, no sequential device is required. The critical path of the whole design corresponds to the path from the input to the output of this module. The QR module design is schematized in Figure 4b.
The block function module receives a 512-bit block of data and a start signal as inputs, and returns another 512-bit block of data and a done signal as outputs. When the start signal is up, this module saves the input block into an initial state register and a current state register, exploits the QR module to compute 20 rounds overwriting the current state register as it goes, then adds the current state to the initial state and outputs the resulting block raising the done signal. This module can integrate a single QR module, requiring four clock cycles per round, or four different QR modules running in parallel, performing an entire round in a single clock cycle, at a higher cost in terms of resource utilization. The block function module design is shown in Figure 4a.
The cipher module has one input and one output port, named *in* and *out*, respectively. A *done* port signals whether output data is returned. The operation to be executed is controlled by the *control* port; the possible commands are listed in Table 1. In general, the cipher module operates on 32-bit words. This module starts the computation of the new ChaCha20 states as soon as the hardware resources are available. For the first word to be encrypted, the encryption operation is "long", as the ChaCha20 state must first be computed; the resulting 16-word state is saved and its first 32 bits are used for the encryption, while at the same time the computation of the next state is started in advance. Then, from the 2nd to the 16th word to be encrypted, the encryption is "short", since the state is already computed and it’s possible to use respectively from the 2nd to the 16th of its 32 bits. At the 17th encryption there are no more state bits available, so the cipher waits for the remaining time of the new state computation; then, the 17th ciphertext word is returned, while at the same time a new state computation is started, and so on. In case less than 32 bits are to be encrypted, padding of the plaintext and then discarding the excess bits of the ciphertext is required.

Table 1: List of control values accepted by the *Cipher* module

Control value	Operation
$i \in \{1, ..., 3\}$	Set *in* as nonce i-th word
$i \in \{4, ..., \mathrm{B}\}$	Set *in* as key $(i-3)$-th word
C	Set *in* as counter initial value
D	Encrpyt/decrypt text *in*
else	No operation

(a) Unrolled design

(b) Pipelined design

Figure 5: Comparison of unrolled and pipelined designs with 4 rounds

3.2. Architectural optimizations

Now we explore three different architectural optimizations: smaller components size, loop unrolling of rounds and computation pipelining.

3.2.1 Sequential design with smaller components

In a sequential design, during each clock cycle only one computation is made. In particular, in our QR module, we integrate a single adder and a single xor bank, which are in turn used. It is reasonable to expect inside the QR module a decrease in occupied resources due to the reduction of integrated components; on the other hand, the sequentiality of the design and the consequent need for memory cells adds complexity to the system. This design choice results in a much shorter critical path, while many clock cycles are required to execute the full quarter-round. Also in this case, both 1-QR and 4-QR variants are evaluated. This design choice allows us to vary the bit size of operands and result of the adder and xor bank modules: smaller components lead to slower computations, as more clock cycles are required, but at the same time the resources utilization is reduced. 32-bit, 16-bit and 8-bit adders and xor banks are explored; as a consequence, a round requires respectively 32, 64, 256 operations in 1-QR variant, and 8, 16, 32 operations in 4-QR variant.

3.2.2 Unrolled design

In an unrolled design, two or more rounds are cascaded in order to compute multiple rounds in one clock cycle, where each round is implemented as four QR modules working in parallel. The number of rounds which are concatenated is called the *unrolling factor*; the ChaCha20 block function has to execute 20 rounds, as a consequence possible unrolling factors are 2, 4, 5, 10 and 20, executed respectively in 10, 5, 4, 2 and 1 clock cycles. The ability to execute multiple rounds in a single clock cycle strongly impacts the critical path of the design, which now corresponds to the path from the input of the first round to the output of the last round. Since now there exists a considerable difference between the maximum clock frequency achievable inside the block function and the one in the rest of the system, we choose to split the design into two clock domains, one for the cipher module, and another one, at a frequency which is much lower and inversely proportional to the unrolling factor, for the block function module. Of course, resource utilization is much higher since more than one round is integrated, and grows linearly with the unrolling factor. Figure 5a schematizes the unrolled ChaCha20 block function module with an unrolling factor of 4.

3.2.3 Pipelined design

The last optimization we explore is the pipelined design, in which the block function is computed using a pipeline. A pipeline is composed by several stages, each stage corresponds to a round function; the number of stages is called the *pipelining factor*. Each stage, except for the first one, receives as inputs the outputs of the previous stage and executes its computation in one clock cycle; the first stage receives the pipeline inputs, while the last stage returns the pipeline outputs. This implies the existence of memory registers between two consecutive rounds which save temporary results. Since the ChaCha20 state is computed in 20 rounds, possible pipelining factors are 2, 4, 5, 10 and 20, executed respectively 10, 5, 4, 2 and 1 times. Since a round is composed by four parallel QR modules, we expect the critical path to remain the QR path, no matter which pipelining factor is selected. We can expect resource utilization to be at least as high as the unrolled designs, since the same number of rounds is integrated, but with a significant increase in FFs utilization, due to the presence of inter-stage registers; of course, such utilization has to grow linearly with the pipelining factor.
The pipeline allows for a form of partial parallelism in the computation of final states: different rounds of different matrices can be computed in parallel at different stages of the pipeline. We exploit this form of parallelism to time overlap the computation of a number of successive states equal to the pipelining factor. Assuming that P_f is the pipelining factor, that i is the initial counter value, and that the j-th state (alternatively referred to as s_j) is the state with the counter value j, during the first clock cycle the cipher passes the i-th initial state to the first stage of the pipeline; in the second clock cycle, while the i-th state has executed its first round, the cipher passes the $i+1$-th initial state to the first stage. The operation is repeated until the pipeline is filled with successive states, precisely the states from i to $i + P_f - 1$. Then, at each clock cycle the pipeline receives as input its own output, forming a closed-loop structure; after the twentieth round of the i-th state being computed the pipeline stops, the computed i-th state is added to the initial state and thus the first 512 bits of plaintext can be encrypted. Each time a state finishes being computed and is ejected from the pipeline, a new successive initial state is inserted into the pipeline, while each state already inside the pipeline executes a round; this happens until all the P_f states computed are used. After that, the whole process is repeated, this time with P_f successive states already entered into the pipeline and partially already processed.
Further expanding the pipeline, an additional variant considered is a 21-stage pipeline, of which the first 20 execute column round and diagonal round alternately, while the 21st stage executes the sum of the 20th round result with the initial state and the XOR combination between the final state and the plaintext. In this way, by integrating the final sum and XOR operations in the pipeline, after the initial pipeline filling stage 512 bits of plaintext can be continuously encrypted with each clock cycle. Moreover, the additional stage does not involve additional resource usage since these are operations necessarily present in any variant; rather, it simplifies of design since the no-longer-needed closed-loop structure is completely eliminated and the entire logic of the block function is moved into the pipeline.
In this specific design the cipher operates on 64-bit words. Figure 5b schematizes the pipelined ChaCha20 block function module with a pipelining factor of 4.

3.3. Side-channel countermeasures

In this subsection we propose side-channel resistant implementations of the ChaCha20 cipher based on masking. In particular, we exploit the threshold implementation and the low cost implementation tecniques.
Since these implementations make use of random numbers, our designs integrate 40-bit Linear-Feedback Shift Registers (LFSR) as random bit generators, each one initialized to a different state and with maximum period. In order to analyze the power consumption, we developed a top-module whose main task is to make the design under test execute multiple encryptions, raising the trigger signal of the oscilloscope during the computations. In detail, this module receives via the UART serial interface nonce, key, counter, a 64-byte plaintext, the number of repetitions of the encryption, a random seed and an additional byte that signals whether the mode is fixed-input or random-input. For each repetition of the encryption, the cipher is set up with nonce, key and counter, a delay of 100 ms is performed to allow time for the oscilloscope to prepare its buffer for the recording, the encryption of the first 4 bytes of the plaintext is executed with trigger up, an additional delay of 1 ms is performed to separate in the power traces the activity of the protected cipher from that of the unprotected top-module, the remaining 60 bytes are then encrypted and, in random-input mode only, the ciphertext is set as new plaintext and its first 4 bytes as new counter. During the last repetition, the top-module transmits back the last ciphertext as proof of good functioning. The finite state machine of the top-module is shown in Figure 6.

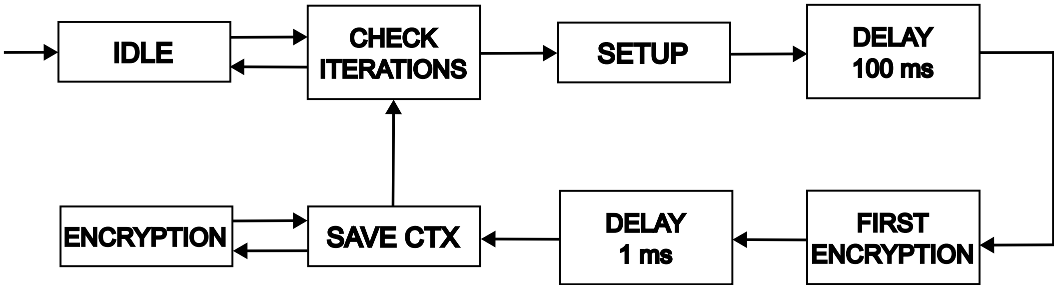


Figure 6: Finite state machine of the top-module designed to manage the protected designs and to allow power measurements

3.3.1 Threshold implementation

In order to build a first-order secure threshold implementation of the cipher, we have to formulate the masking schemes of its two arithmetic functions, i.e., xor and modular addition. In particular, our designs integrate 32-bit ripple-carry adders, which are composed of 32 full adders, each of which receives two operands a, b and a carry-in c_i as inputs and returns a result $r = a \oplus b \oplus c_i$ and a carry-out $c_{i+1} = (a \wedge b) \vee (a \wedge c_i) \vee (b \wedge c_i)$ for each bit i from 0 to 31, given $c_0 = 0$. As a consequence, we have to develop protected implementations of the fundamental 1-bit functions xor and carry-out.
We start from the xor function $r = a \oplus b \oplus c$.
We split each of the three operands a, b, c into three shares:

$$\begin{aligned} r &= \sum_{i=0}^{2} a_i \oplus \sum_{i=0}^{2} b_i \oplus \sum_{i=0}^{2} c_i \\ &= a_0 \oplus a_1 \oplus a_2 \oplus b_0 \oplus b_1 \oplus b_2 \oplus c_0 \oplus c_1 \oplus c_2 \end{aligned} \tag{5}$$

In order to guarantee the non-completeness property, we also partition the result r into three shares in such a way that the i-th share of the result does not depend on the i-th share of any operand:

$$\begin{cases} r_0 = a_1 \oplus b_2 \oplus c_1 \\ r_1 = a_2 \oplus b_0 \oplus c_2 \\ r_2 = a_0 \oplus b_1 \oplus c_0 \end{cases} \tag{6}$$

We have thus obtained the scheme of our threshold implementation of the 3-operand xor function; it is trivial to obtain the 2-operand xor function by removing the c operand.
Similar work is done for the carry-out function $r = ab \vee bc \vee ca$.
We make the function employ only logical conjunctions and negations, applying De Morgan's laws:

$$\begin{aligned} r &= \overline{(\overline{ab})(\overline{bc})} \vee ca \\ &= \overline{(\overline{ab})\ (\overline{bc})\ (\overline{ca})} \\ &= 1 \oplus ((1 \oplus ab)(1 \oplus bc)(1 \oplus ca)) \end{aligned} \tag{7}$$

We split the three operands a, b, c into three shares:

$$\begin{aligned} r &= 1 \oplus ((1 \oplus \sum_{i=0}^{2} a_i \sum_{i=0}^{2} b_i)(1 \oplus \sum_{i=0}^{2} b_i \sum_{i=0}^{2} c_i)(1 \oplus \sum_{i=0}^{2} c_i \sum_{i=0}^{2} a_i)) \\ &= \sum_{i=0}^{2} \sum_{j=0}^{2} a_i b_j \oplus \sum_{i=0}^{2} \sum_{j=0}^{2} b_i c_j \oplus \sum_{i=0}^{2} \sum_{j=0}^{2} c_i a_j \end{aligned} \tag{8}$$

We partition the result r into three shares the same way as before:

$$\begin{cases} r_0 = a_1 b_2 \oplus a_2 b_1 \oplus a_1 c_2 \oplus a_2 c_1 \oplus b_1 c_2 \oplus b_2 c_1 \oplus a_1 b_1 \oplus b_2 c_2 \oplus c_1 a_1 \\ r_1 = a_0 b_2 \oplus a_2 b_0 \oplus a_0 c_2 \oplus a_2 c_0 \oplus b_0 c_2 \oplus b_2 c_0 \oplus a_2 b_2 \oplus b_0 c_0 \oplus c_2 a_2 \\ r_2 = a_1 b_0 \oplus a_0 b_1 \oplus a_1 c_0 \oplus a_0 c_1 \oplus b_1 c_0 \oplus b_0 c_1 \oplus a_0 b_0 \oplus b_1 c_1 \oplus c_0 a_0 \end{cases} \tag{9}$$

We have now obtained the scheme of our threshold implementation of the carry-out function.
Finally, since the cipher's internal encoding is now different from the external encoding, the input data must be translated into the three-shares encoding just before entering the cipher, while the output data must be converted back into the normal encoding just before exiting the cipher. For this purpose, encoders are placed in the input interface and decoders in the output interface.
The encoder function of bit b employs random bits in order to guarantee uniformity:

$$\begin{cases} b_0 = b \oplus z_0 \oplus z_1 \\ b_1 = z_0 \\ b_2 = z_1 \end{cases} \tag{10}$$

The decoder function sums the three shares obtanining the bit $b = b_0 \oplus b_1 \oplus b_2$.
We now prove the security properties of Equations 6, 9, 10 by recomposing the three shares of each input bit:

$$\begin{aligned} r &= a_0 \oplus a_1 \oplus a_2 \oplus b_0 \oplus b_1 \oplus b_2 \oplus c_0 \oplus c_1 \oplus c_2 \\ &= (a_0 \oplus a_1 \oplus a_2) \oplus (b_0 \oplus b_1 \oplus b_2) \oplus (c_0 \oplus c_1 \oplus c_2) \\ &= a \oplus b \oplus c \end{aligned} \tag{11}$$

In the same way we prove the correctness property of the carry-out function:

$$\begin{aligned} r = {} & a_1b_2 \oplus a_2b_1 \oplus a_1c_2 \oplus a_2c_1 \oplus b_1c_2 \oplus b_2c_1 \oplus a_1b_1 \oplus b_2c_2 \oplus c_1a_1 \\ & \oplus a_0b_0 \oplus a_0b_2 \oplus a_2b_0 \oplus a_0c_2 \oplus a_2c_0 \oplus b_0c_2 \oplus b_2c_0 \oplus a_2b_2 \oplus b_0c_0 \\ & \oplus c_2a_2 \oplus b_1c_1 \oplus c_0a_0 \oplus a_1b_0 \oplus a_0b_1 \oplus a_1c_0 \oplus a_0c_1 \oplus b_1c_0 \oplus b_0c_1 \\ = {} & (a_0b_0 \oplus a_0b_1 \oplus a_0b_2 \oplus a_1b_0 \oplus a_1b_1 \oplus a_1b_2 \oplus a_2b_0 \oplus a_2b_1 \oplus a_2b_2) \\ & \oplus (a_0c_0 \oplus a_0c_1 \oplus a_0c_2 \oplus a_1c_0 \oplus a_1c_1 \oplus a_1c_2 \oplus a_2c_0 \oplus a_2c_1 \oplus a_2c_2) \\ & \oplus (c_0b_0 \oplus c_0b_1 \oplus c_0b_2 \oplus c_1b_0 \oplus c_1b_1 \oplus c_1b_2 \oplus c_2b_0 \oplus c_2b_1 \oplus c_2b_2) \\ = {} & ab \oplus ac \oplus cb \end{aligned} \tag{12}$$

It is easy to observe the correctness of the encoder function $b = b \oplus z_0 \oplus z_1 \oplus z_0 \oplus z_1$.
We now prove the uniformity property. Firstly we demonstrate that the use of random bits in the encoder function, as represented in Equation 10, guarantees the uniformity property for encoder output shares. We consider a generic non-uniform bit x as input of our encoder function:

$$\begin{cases} r_0 = x \oplus z_0 \oplus z_1 \\ r_1 = z_0 \\ r_2 = z_1 \end{cases} \tag{13}$$

r_1 and r_2 are random bits by definition, we compute the distribution of r_0:

$$\begin{aligned} \Pr(r_0) &= \Pr(x \oplus z_0 \oplus z_1 = 1) \\ &= \Pr(x \oplus z_2 = 1) \\ &= \Pr(x = 1 \wedge z_2 = 0) + \Pr(x = 0 \wedge z_2 = 1) \\ &= \Pr(x = 1) \cdot \Pr(z_2 = 0) + \Pr(x = 0) \cdot \Pr(z_2 = 1) \\ &= 0.5 \cdot (\Pr(x = 1) + \Pr(x = 0)) \\ &= 0.5 \end{aligned} \tag{14}$$

No matter which value x is represented, its shares are uniformly distributed.
Secondly, we demonstrate that the xor and carry-out functions described respectively in Equations 6, 9 are uniform. As explained in [4], a masked function $\mathbf{f}$ with s_{in} input shares and s_{out} output shares, threshold implementation of function f, is uniform if and only if

$$N = |\{\mathbf{x} \in \mathrm{Sh}(x) \,|\, \mathbf{f}(\mathbf{x}) = \mathbf{y}\}| = \frac{2^{n(s_{in}-1)}}{2^{m(s_{out}-1)}} \tag{15}$$

for each input $x \in \mathbb{F}^n$, $y = f(x) \in \mathbb{F}^m$ and $\mathbf{y} \in \mathrm{Sh}(y)$, given $\mathrm{Sh}(z)$ the set of correct sharings of variable z, n the function input bits and m the function output bits.
We can check the uniformity of our proposed function implementations by computing their uniformity tables. A uniformity table of a masked function $\mathbf{f}$ shows, for each unshared input and for each possible output sharing, the number of input sharings that generate the output sharing. In our threshold implementation case $s_{in} = s_{out} = 3$,

so a function is uniform if, for each input, each correct output sharing occurs $N = 2^{2(n-m)}$ times, while the other output sharings never occur.
In Tables 2, 3, 4 we show the uniformity tables of the 2-operand xor, 3-operand xor and carry-out functions. Note that in all our functions m is equal to 1, while n is equal to 2, 3 and 3, respectively; this means that N is equal to 4, 16 and 16.

Table 2: Xor-2 function uniformity table, a, b are operands, r_0, r_1, r_2 are output shares

$(\mathbf{a}, \mathbf{b})$	$\mathbf{r_0\ r_1\ r_2}$							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
(0, 0)	4	0	0	4	0	4	4	0
(0, 1)	0	4	4	0	4	0	0	4
(1, 0)	0	4	4	0	4	0	0	4
(1, 1)	4	0	0	4	0	4	4	0

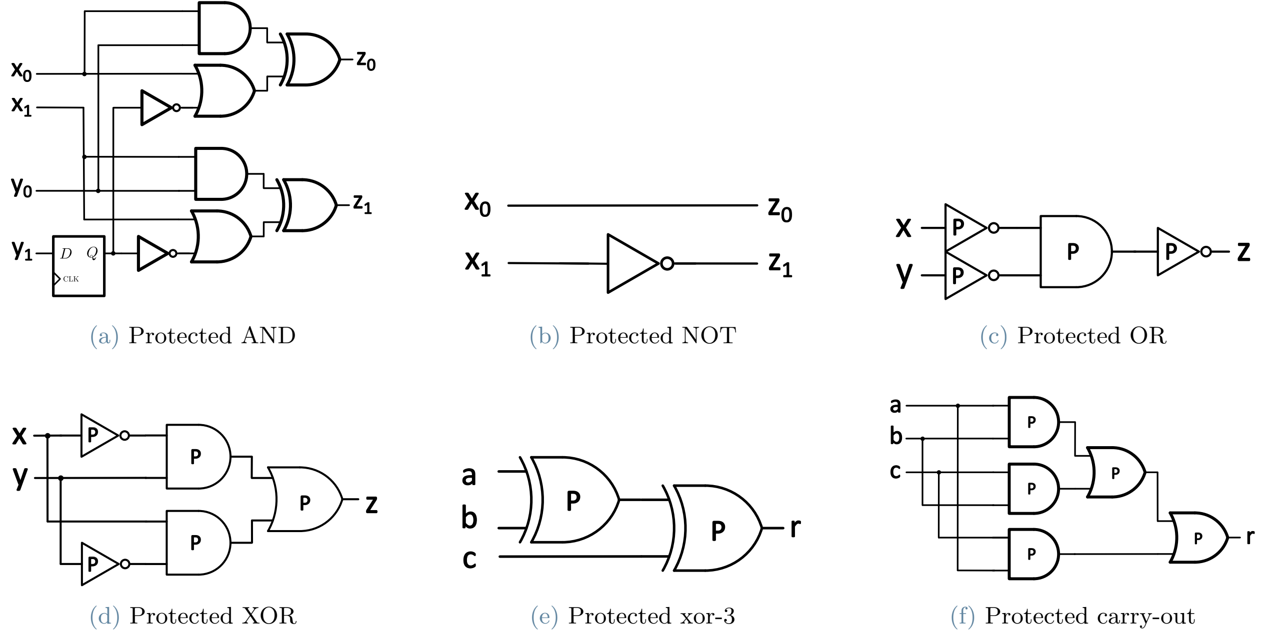
Table 3: Xor-3 function uniformity table, a, b, c are operands, r_0, r_1, r_2 are output shares

$(\mathbf{a}, \mathbf{b}, \mathbf{c})$	$\mathbf{r_0\ r_1\ r_2}$							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
(0, 0, 0)	16	0	0	16	0	16	16	0
(0, 0, 1)	0	16	16	0	16	0	0	16
(0, 1, 0)	0	16	16	0	16	0	0	16
(0, 1, 1)	16	0	0	16	0	16	16	0
(1, 0, 0)	0	16	16	0	16	0	0	16
(1, 0, 1)	16	0	0	16	0	16	16	0
(1, 1, 0)	16	0	0	16	0	16	16	0
(1, 1, 1)	0	16	16	0	16	0	0	16

Table 4: Carry-out function uniformity table, a, b, c are operands, r_0, r_1, r_2 are output shares

$(\mathbf{a}, \mathbf{b}, \mathbf{c})$	$\mathbf{r_0\ r_1\ r_2}$							
	000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
(0, 0, 0)	16	0	0	16	0	16	16	0
(0, 0, 1)	16	0	0	16	0	16	16	0
(0, 1, 0)	16	0	0	16	0	16	16	0
(0, 1, 1)	0	16	16	0	16	0	0	16
(1, 0, 0)	16	0	0	16	0	16	16	0
(1, 0, 1)	0	16	16	0	16	0	0	16
(1, 1, 0)	0	16	16	0	16	0	0	16
(1, 1, 1)	0	16	16	0	16	0	0	16

We now illustrate that in the proposed masking schemes respect the non-completeness property, i.e., that each output share is indipendent from exactly one input share. In Table 5, share dependencies are reported for xor and carry-out functions.



(a) Protected AND (b) Protected NOT (c) Protected OR

(d) Protected XOR (e) Protected xor-3 (f) Protected carry-out

Figure 7: LC implementation of logic functions, **P** gates denote protected gates

Table 5: Shares dependencies for xor-2, xor-3 and carry-out functions, a_i, b_i, c_i are i-th shares of operands a, b, c, r_i is i-th share of result r

Share	Xor-2	Xor-3	Carry-out
r_0	$a_1,\ b_2$	$a_1,\ b_2,\ c_1$	$a_1,\ a_2,\ b_1,\ b_2,\ c_1,\ c_2$
r_1	$a_2,\ b_0$	$a_2,\ b_0,\ c_2$	$a_0,\ a_2,\ b_0,\ b_2,\ c_0,\ c_2$
r_2	$a_0,\ b_1$	$a_0,\ b_1,\ c_0$	$a_0,\ a_1,\ b_0,\ b_1,\ c_0,\ c_1$

3.3.2 Low cost implementation

As a term of comparison for the proposed threshold implementation, we now consider the low cost implementation of the AND gate, and based on it we build all the useful logic functions.
The NOT gate is built simply by flipping one of the two share bits.
From the implementations of the AND and NOT gates, we exploit De Morgan's laws in order to build the OR gate $z = x \vee y = \neg((\neg x) \wedge (\neg y))$.
From the implementations of the AND, NOT and OR gates, we build the XOR gate $z = x \oplus y = (\neg x \wedge y) \vee (x \wedge \neg y)$.
On the basis of these logic gates it is trivial to build the usual xor function $r = (a \oplus b) \oplus c$ and carry-out function $r = ((a \wedge b) \vee (b \wedge c)) \vee (c \wedge a)$.
Finally, we consider encoders and decoders. A decoder as usual recomposes a bit by summing its two shares, while an encoder receives bit b and decomposes it using a random bit, as follows:

$$\begin{cases} b_0 = b \oplus z \\ b_1 = z \end{cases} \tag{16}$$

In Figure 7 the logic design of protected functions are shown, while in Table 6 evaluation times are listed.

Table 6: Evaluation times of LC implementation of logic functions

Logic	Clock cycles
AND	1
OR	1
XOR	2
carry-out	3
xor-3	4

4. Experimental evaluation

The following are the implementation details of the various variants of the ChaCha20 cipher. The hardware description language used to design the various implementations is SystemVerilog, the software used to synthesize those designs is Vivado Design Suite. The target FPGA is an Artix-7 XC7A100TCSG324-1 device. For synthesis and implementations, the Vivado default strategy for unprotected versions and the *Flow_RuntimeOptimized* strategy for protected versions were used. The maximum frequencies were obtained by constraining the design to a clock period, synthesizing and implementing it, and then repeatedly constraining it to a new clock period equal to the previous clock period minus the Worst Negative Slack obtained, as long as the time constraint can be met. For the power analysis, on the FPGA side the previously described top module was used as the manager of the design under test, while on the host side a python script was used to transmit and receive data via serial port; measurements were made with an oscilloscope with an electromagnetic probe.

4.1. Figures of merit

In Table 7 timing and slice logic distribution data is reported.

Table 7: Resources and timing values; Slices, Look-Up Tables (LUTs), Flip-Flops (FFs) and Clock frequency are reported.

Design	Adder width	QRs	Round/ Stages	Slices	LUTs	FFs	Freq. (MHz)
Sequential	8-bit	1	-	912	2579	2860	211
	16-bit	1	-	923	2583	3086	199
	32-bit	1	-	927	2645	3587	208
	8-bit	4	-	1117	3220	3819	210
	16-bit	4	-	1142	3302	4133	215
	32-bit	4	-	1192	3138	4756	212
Combinatorial	32-bit	1	-	776	2122	2283	72
		4	-	953	2483	2285	77
Unrolled	32-bit	-	2	1024	2971	3206	40
		-	4	1580	4014	3189	20
		-	5	1944	5813	3224	15
		-	10	2509	8121	3207	7
		-	20	4083	14543	3207	4
Pipelined	32-bit	-	2	1119	3821	2513	66
		-	4	1752	6273	3553	72
		-	5	2068	7385	4078	67
		-	10	3619	13358	6675	67
		-	20	6760	25036	11755	71
		-	21	4363	16318	11150	88
TI combinatorial	32-bit	1	-	5207	16054	16310	164
		4	-	5717	17479	18977	171
LC combinatorial	32-bit	1	-	4648	13802	17120	180
		4	-	5165	16055	20707	169

In Table 8 performance data is reported. In all designs, the encryption time varies with the need to compute a new keystream, generating a periodic behavior. The cycles/byte ratio is calculated on blocks of text covering one or more exact periods: in this way, each time cost is amortized over all clock cycles, so this value is representative of the general performance of the design. Specifically, it is calculated for pipelined versions on a 1280-byte text, for other versions on a 64-byte text, as:

$$Cycles/byte = \frac{\text{Clock cycles required}}{\text{Plaintext bits}} \tag{17}$$

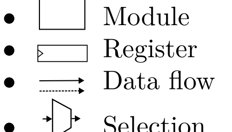
Given the cycles/byte value, throughput is calculated as:

$$Throughput = \frac{8}{\text{Minimum clock period} \times \text{Cycles/byte}} \tag{18}$$

Table 8: Performance values; Cycles/byte, Throughput and Throughput/area are reported.

Design	**Adder width**	**QRs**	**Round/ Stages**	**Cycles/byte (cycle/B)**	**Thr. (Mb/s)**	**Thr./slice (Mb/s/slice)**
Sequential	8-bit	1	-	46.24	36.50	0.04
	16-bit	1	-	26.32	60.41	0.07
	32-bit	1	-	16.29	102.29	0.11
	8-bit	4	-	11.66	143.73	0.13
	16-bit	4	-	6.62	259.73	0.23
	32-bit	4	-	4.11	412.06	0.35
Combinatorial	32-bit	1	-	1.30	444.32	0.57
		4	-	0.36	1708.70	1.79
Unrolled	32-bit	-	2	0.69	1832.44	1.79
		-	4	0.38	1693.10	1.07
		-	5	0.63	1505.88	0.77
		-	10	0.75	1254.94	0.50
		-	20	1.00	1052.63	0.26
Pipelined	32-bit	-	2	0.28	1788.14	1.60
		-	4	0.20	2815.83	1.61
		-	5	0.19	2946.62	1.42
		-	10	0.16	3618.12	1.00
		-	20	0.14	3935.05	0.58
		-	21	0.13	5624.40	1.29
TI combinatorial	32-bit	1	-	178.56	7.34	< 0.01
		4	-	45.44	30.11	0.01
LC combinatorial	32-bit	1	-	535.37	2.69	< 0.01
		4	-	136.00	9.95	< 0.01

In Figures 8, 9, 10, 11, 12, 13, 14, 15, 16 critical paths are highlighted in red on the block diagrams representing the main data flows inside the various designs. Regarding combinatorial and sequential variants, the critical paths of 1-QR versions are shown, while, regarding the unrolled and pipelined variants, the critical paths of the 2-round versions are presented, although the other versions not shown are similar and easily deducible. In Figures 15, 16 random bits line is highlighted in green; for simplicity the diagram omits the encoders and decoders in the I/O interface of the *Cipher* module; the former are connected to the random bits line. Here are the symbols presented:



- Module
- Register
- Data flow
- Selection

- Addressing
- Adder module
- XOR module

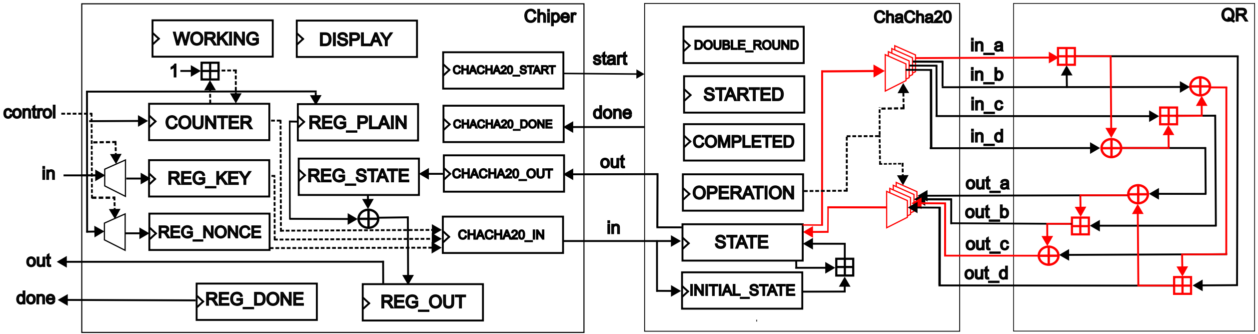


Figure 8: Combinatorial design critical path

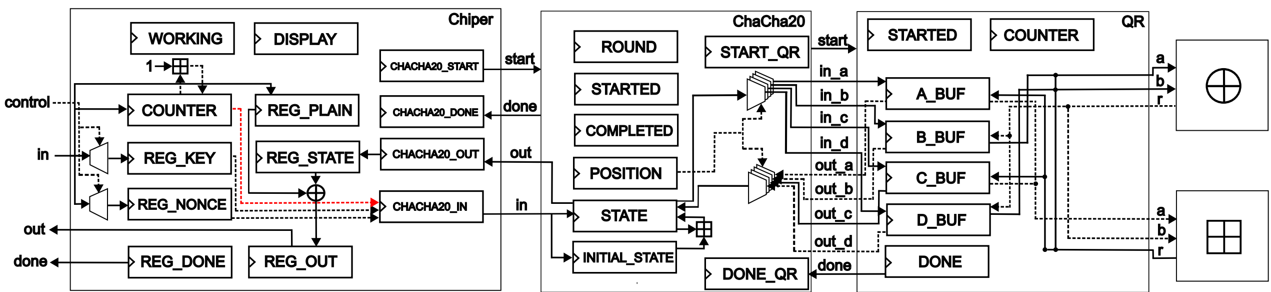


Figure 9: 32-bit sequential design critical path

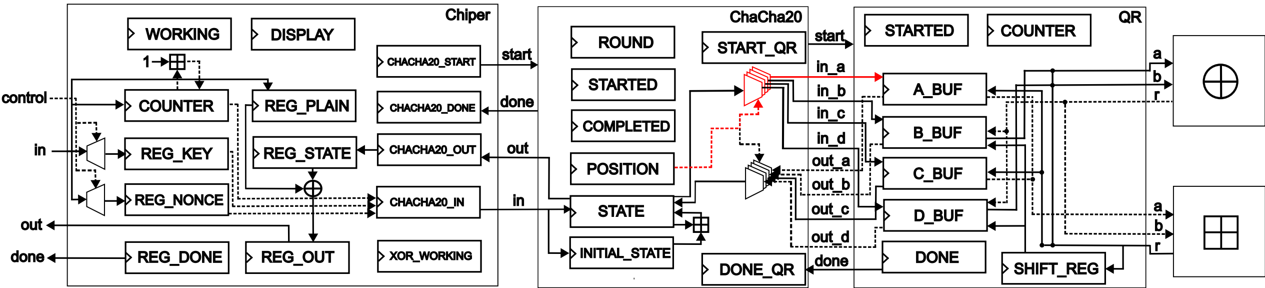


Figure 10: 16-bit sequential design critical path

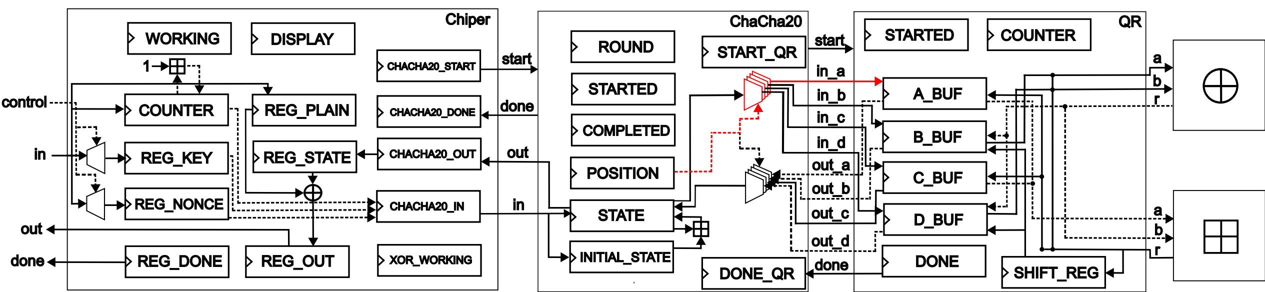


Figure 11: 8-bit sequential design critical path

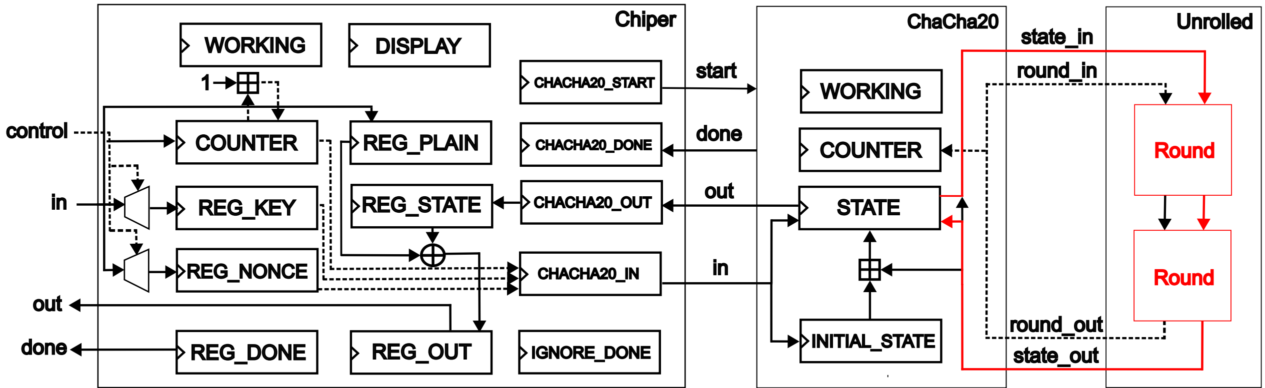


Figure 12: 2-round unrolled design critical path

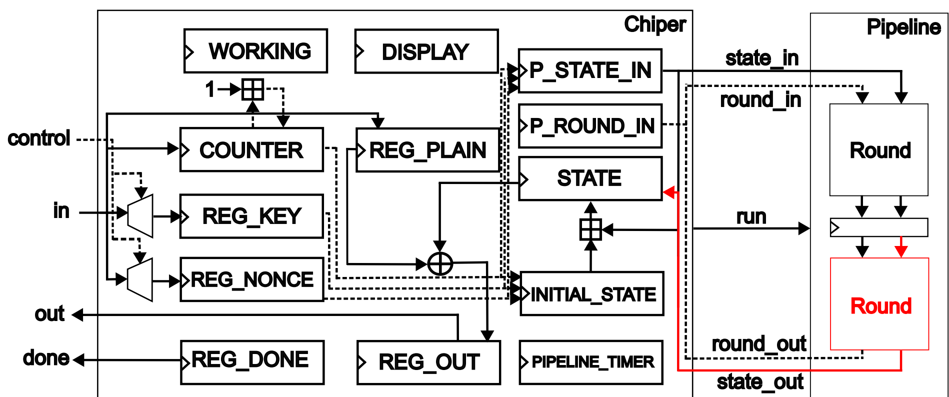


Figure 13: 2-stage pipelined design critical path

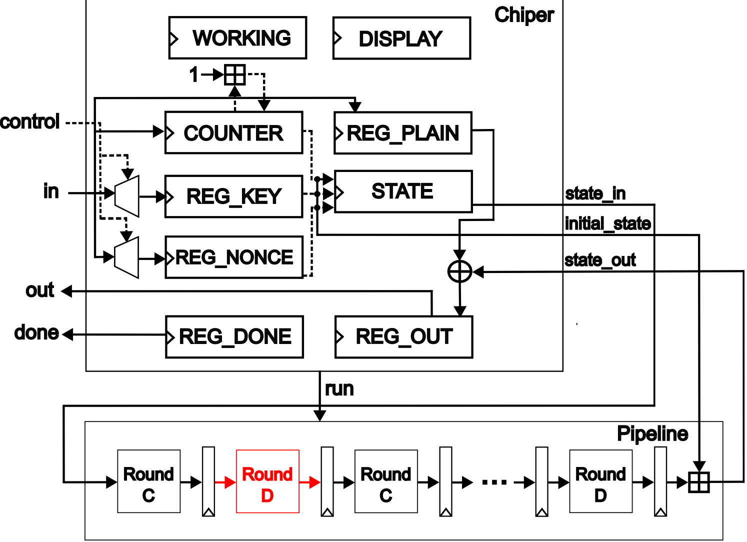


Figure 14: 21-stage pipelined design critical path

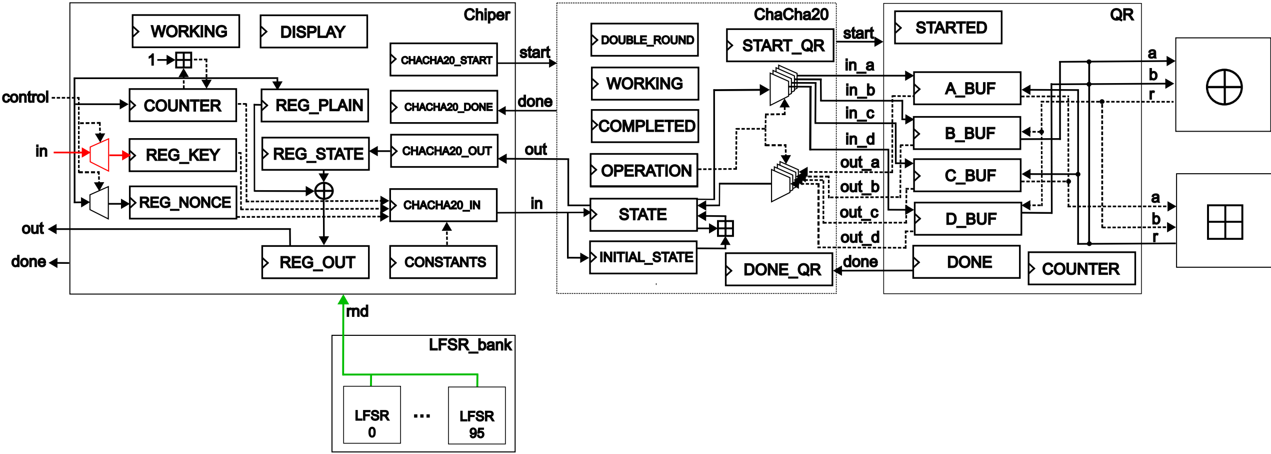


Figure 15: TI combinatorial design critical path

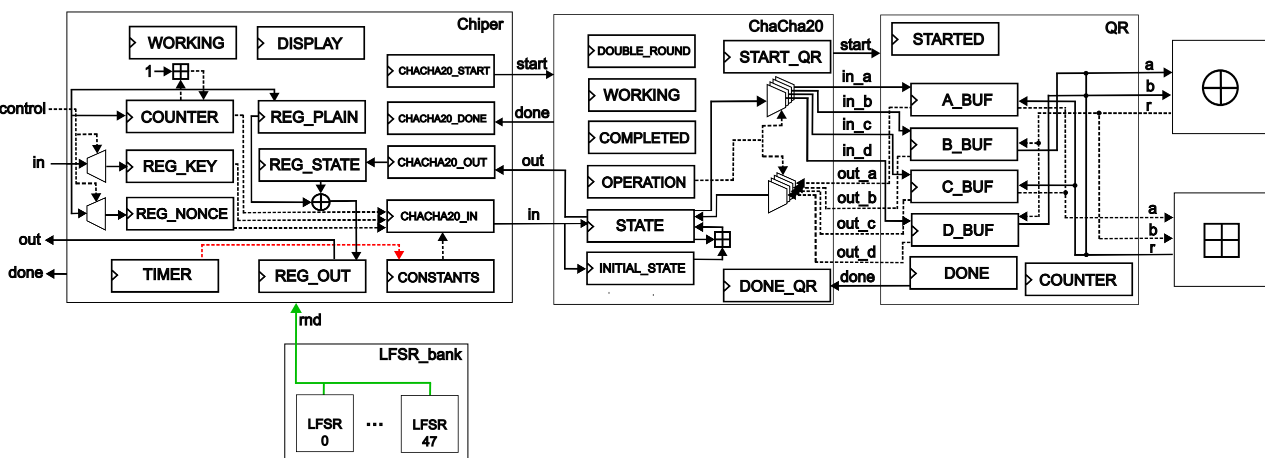


Figure 16: LC combinatorial design critical path

We now draw an overview of the collected data. The first observation concerns the comparison between the 1-QR and 4-QR variants of each version of the cipher. On average, the 4-QR variants occupy approximately 25% more slices in the unprotected versions, and about 10% in the protected versions. On the other hand, the resulting performance is 3.6 to 4.7 times that of the respective 1-QR variants, as we expected.
Regarding sequential designs, the data confirm our expectations of higher speed and cost as the bit size of the components increases; doubling the size, on average, we get performance jumps of 68% and slices occupancy jumps of 2%, achieving an encryption rate of 412 Mb/s. On the other hand, combinatorial designs prove to be better in almost absolute terms: the 1-QR and 4-QR variants perform 8% and as much as 315% better, respectively, than the fastest sequential design, achieving about an encryption rate of 1.7 Gb/s, while at the same time occupying 15% less and just 5% more slices than the least expensive sequential design. Clearly, the simplicity of a fully combinatorial design causes large resource savings, which even the use of small bit size components cannot concretely achieve.
Moving to multi-round designs, unrolled designs have larger resource utilization: compared with the combinatorial 4-QR design, they increase the number of occupied slices from 7% with an unrolling factor of 2, to as much as 328% with an unrolling factor of 20. The 2-round variant beats the best solution so far in terms of throughput, obtaining a data rate of 1.8 Gb/s; as the unrolling factor increases, throughput decreases. Pipelined designs tend to have even higher resource utilization, but in this case performance is directly proportional to the pipelining factor, and acheives a thoughput of 5.6 Gb/s with 4363 slices in the 21-stage variant, which is the best result found in the entire exploration.
Finally, we examine the data of the protected designs. Of course, we expect a significant increase in resource occupancy, given the much greater design complexity due to the decomposition of each bit into three shares, the resulting multiplication of required logic and memory, as well as the generation of pseudorandom bits; we also

expect a strong reduction in performance, since every intermediate result must be saved into registers before being used, limiting the number of operations to one per clock cycle. Looking at the data, it is clear that the protected designs are not comparable to the unprotected ones: the fastest protected design is slower than the slowest unprotected design, while the resource occupancy data are comparable to those of the pipelined design with 20/21 stages. Compared to the low-cost variant, the proposed threshold implementation has a significantly higher encryption speed, which, considering the 4-QR versions, is 3 times, achieving 30.11 Mb/s; on the other hand, low-cost variant saves 10% of occupied slices.
In Figures 17, 18, 19, 20 comparisons between cipher versions are reported. Unprotected versions are colored blue, while protected versions are colored red.

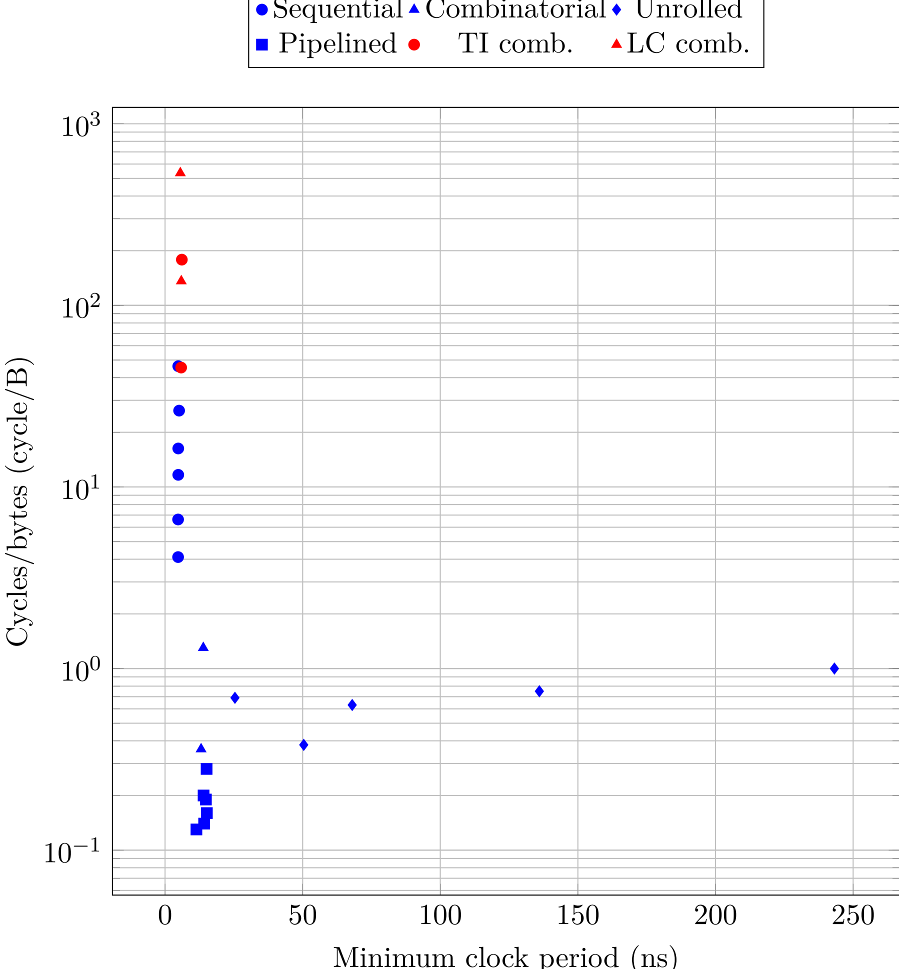


Figure 17: Minimum clock period - cycles/byte comparison

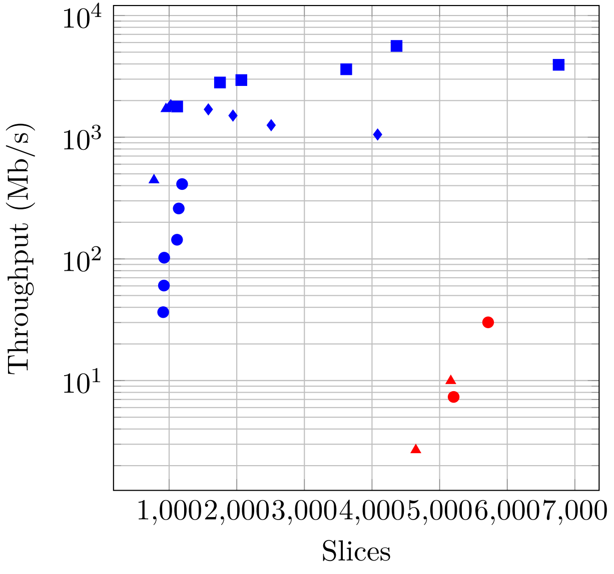


Figure 18: Slices - Thr. comparison

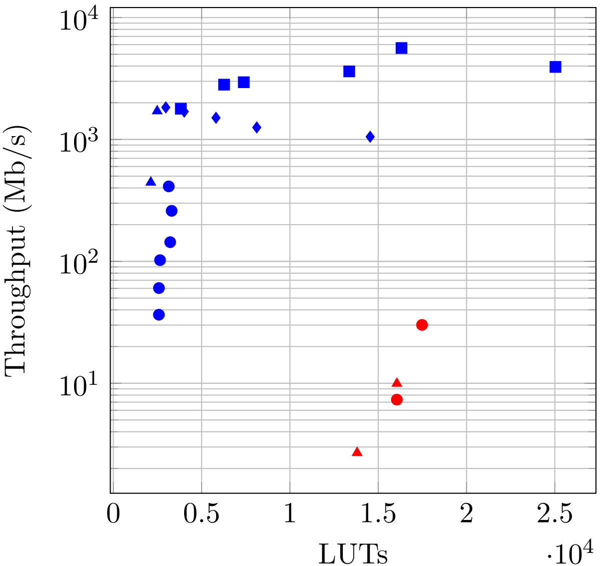


Figure 19: LUTs - Thr. comparison

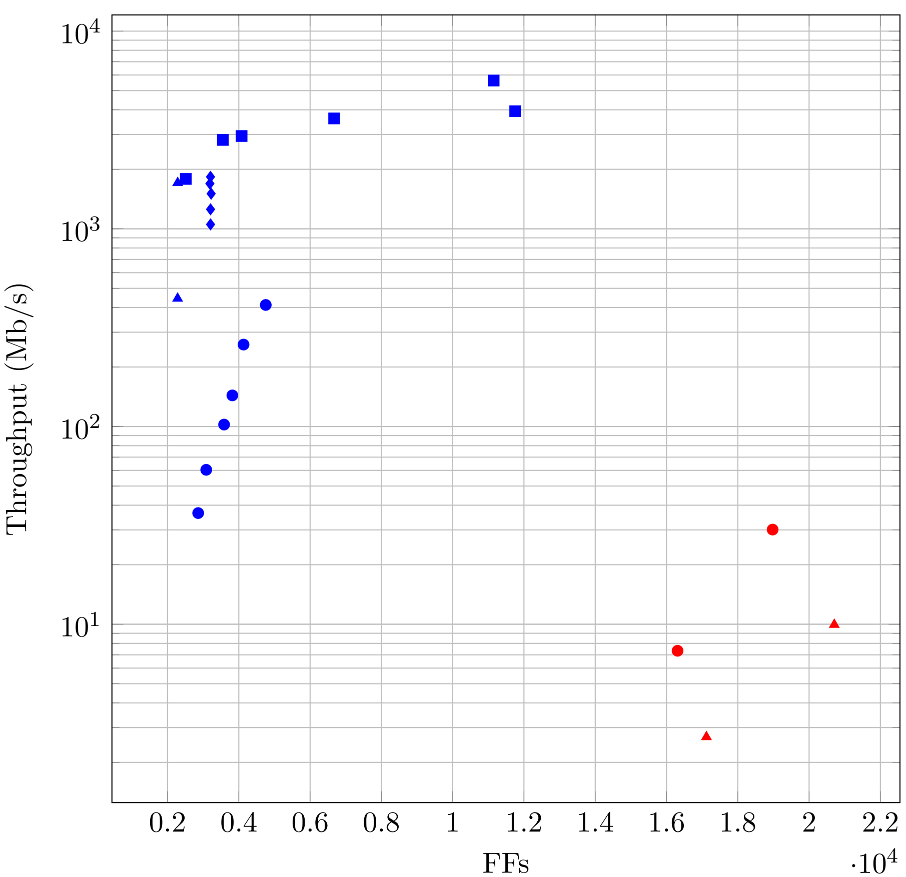
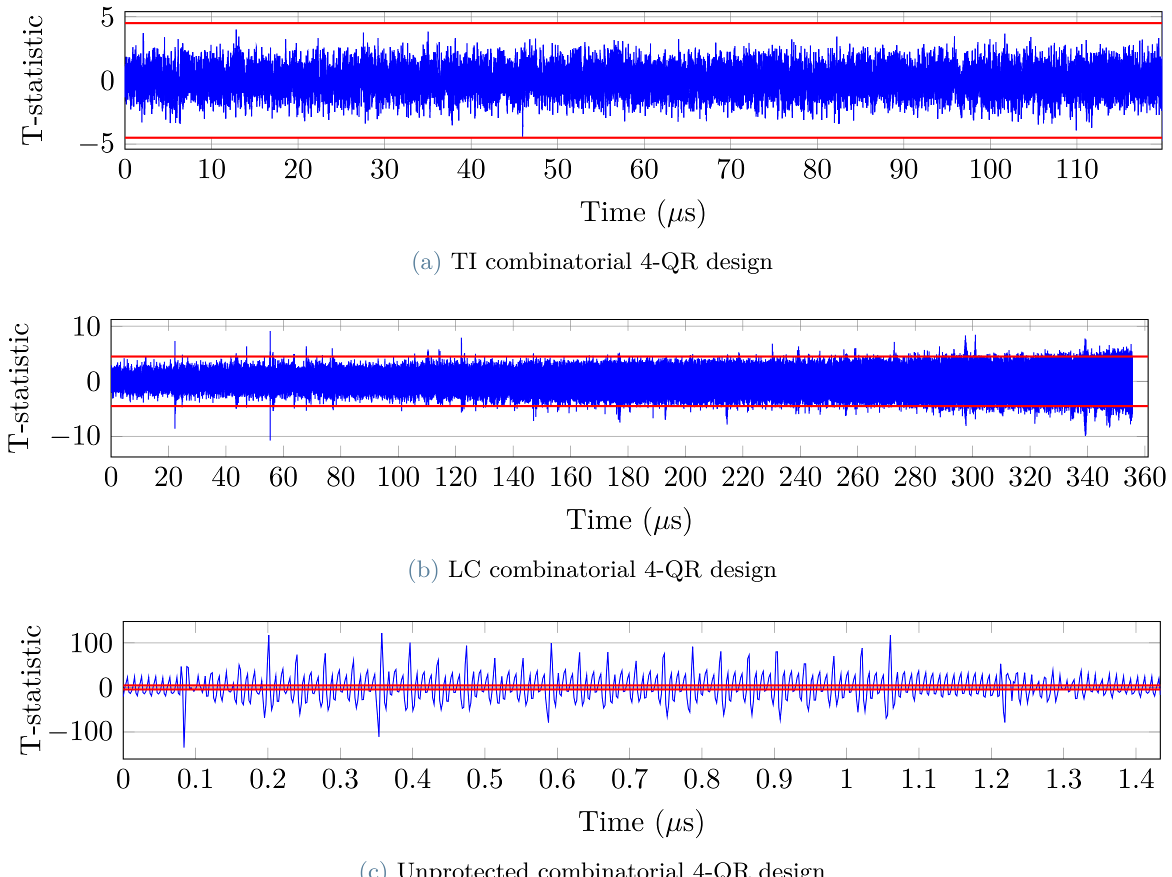


Figure 20: FFs - Throughput comparison

4.2. Power analysis

Through TVLA methodology we validate the side-channel resistance of our proposed TI countermeasure, and compare it with the LC countermeasure and with the absence of protections. For this purpose, we considered the combinatorial 4-QR design, analyzing its power consumption in those three protection arrangements. Each test is conducted by collecting 100k power traces. To establish with 99.999% confidence that the design is safe, the t-statistic must be within the range ± 4.5, however a small margin of overrun can be tolerated. In Figure



(a) TI combinatorial 4-QR design

(b) LC combinatorial 4-QR design

(c) Unprotected combinatorial 4-QR design

Figure 21: Results of TVLA t-tests with 100k traces during the computation of the ciphertext; red lines represent the threshold of ±4.5

21 the results of this analysis are shown. Our proposed TI countermeasure remains within the predetermined range in each point in time, while in the unprotected design the threshold is largely exceeded throughout the calculation, with the t-statistic reaching an absolute value of 135; unexpectedly, even in the LC countermeasure the crossing of the threshold occurs in some points in time, reaching an absolute value of 11.6.

4.3. Comparison with other works

Comparisons between FPGA implementations are always difficult, as the figures of merit depend strictly on the hardware target, which is often different between works. We try to compare our proposed designs with previous works about implementations of ChaCha20, and also with the current standard for symmetric cryptography AES. In particular, as a term of comparison, among our designs we choose the one with the best throughput, which is the 21-stage pipelined variant, and the one with the best throughput - slices ratio, which is the 4-QR combinatorial design. In Table 9 we compare our proposals with the existing hardware implementations of ChaCha20, while in Table 10 we do the same with the existing implementations of AES.
In terms of performance, the comparison shows that our pipelined design beats all existing ChaCha20 and AES implementations for the FPGA family we use, Artix-7. In addition, it beats all existing ChaCha20 implementations except [14], which, however, is not an FPGA-targeted implementation but a VLSI implementation. Finally, in terms of performance per area, our combinatorial design beats all Artix-7, Spartan-II, Spartan-III and Virtex AES implementations.

Table 9: Comparison with existing ChaCha20 implementations, sorted by target and throughput

Impl.	Target	Slices	BRAMs	LUTs	FFs	Freq. (MHz)	Thr. (Gb/s)	Thr./Area (Gb/s/slice)
[7]	Artix-7	-	0	14110	13961	115	0.71	-
Comb 4-QR	Artix-7	953	0	2483	2285	77	1.71	1.79
[32]	Artix-7	1076	0	3837	1949	100	2.23	2.07
Pipeline-21	Artix-7	4363	0	16318	11150	88	5.62	1.29
[34]	UltraScale+	< 1923	-	< 4000	-	-	0.50	> 0.26
[2]	Virtex-6	49	2	-	-	362	0.27	5.43
[29]	Virtex-7	3034	0	10808	3731	166	0.95	0.31
[25]	Virtex-7	680	4	1387	2210	375	2.24	3.37
[25]	Virtex-7	852	0	2392	2873	394	2.35	2.82
[14]	180 nm	-	-	-	-	132	6.80	-

Table 10: Comparison with existing AES implementations, sorted by target and throughput

Impl.	Target	Slices	BRAMs	LUTs	FFs	Freq. (MHz)	Thr. (Gb/s)	Thr./Area (Gb/s/slice)
[16]	Artix-7	554	76	-	-	177	0.12	0.21
Comb 4-QR	Artix-7	953	0	2483	2285	77	1.71	1.79
[28]	Artix-7	4300	40	-	-	-	4.00	0.93
Pipeline-21	Artix-7	4363	0	16318	11150	88	5.62	1.29
[12]	Spartan-II	124	2	-	-	67	< 0.01	0.01
[5]	Spartan-II	222	3	-	-	60	0.17	0.75
[39]	Spartan-III	25107	0	-	-	196	25.00	1.00
[12]	Spartan-III	17425	0	-	-	196	25.10	1.44
[40]	Spartan-6	3788	0	-	-	232	29.73	7.84
[10]	Spartan-6	-	0	9375	256	887	113.50	-
[8]	Virtex	10992	0	-	-	32	1.94	0.18
[6]	Virtex	12600	0	-	-	130	12.20	0.97
[37]	Virtex-II	3766	100	-	-	179	22.93	6.09
[33]	Virtex-II Pro	20249	0	-	-	151	19.28	0.95
[36]	Virtex-II Pro	6541	0	-	-	222	29.77	4.55
[1]	Virtex-4	1132	0	-	-	112	14.38	12.71
[9]	Virtex-4	3425	0	5271	2132	576	73.73	21.53
[21]	Virtex-5	4879	0	14210	18542	733	93.85	19.24
[15]	Virtex-5	14799	0	-	-	233	119.30	8.06
[38]	Virtex-6	2252	244	-	-	471	60.29	26.77
[22]	Virtex-6	3900	0	-	-	573	73.39	18.81
[30]	Virtex-6	10280	0	-	-	764	97.80	9.51
[30]	Virtex-6	35328	0	-	-	508	260.15	7.36
[28]	Virtex-7	5145	40	-	-	-	16.00	3.11

5. Conclusions

In this thesis we proposed an hardware implementation of the cipher based on fully-combinatorial quarter-round units, exploring three different architectural optimizations, i.e., smaller component size, loop unrolling and pipelining. We explored a countermeasure from side-channel attacks based on the threshold implementation tecnique proposing masking schemes for its arithmetic components, then we compared it with an existing gate-

level masking countermeasure. We reported the figures of merit of these implementations targeting an Artix-7 FPGA, reaching 5.62 Gb/s with a pipelined design and 1.79 Mb/s/slice with a combinatorial design. We evaluated the security of protected designs by means of statistical power analysis.

References

[1] C Arul Murugan, P Karthigaikumar, and Sridevi Sathya Priya. FPGA implementation of hardware architecture with AES encryptor using sub-pipelined S-box techniques for compact applications. *Automatika: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije*, 61(4):682–693, 2020.

[2] Nuray At, Jean-Luc Beuchat, Eiji Okamoto, Ismail San, and Teppei Yamazaki. Compact hardware implementations of chacha, blake, threefish, and skein on fpga. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(2):485–498, 2013.

[3] Daniel J Bernstein et al. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer, 2008.

[4] Tim Beyne and Begül Bilgin. Uniform first-order threshold implementations. In *Selected Areas in Cryptography–SAC 2016: 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers 23*, pages 79–98. Springer, 2017.

[5] Paweł Chodowiec and Kris Gaj. Very compact FPGA implementation of the AES algorithm. In *International workshop on cryptographic hardware and embedded systems*, pages 319–333. Springer, 2003.

[6] Pawel Chodowiec, Po Khuon, and Kris Gaj. Fast implementations of secret-key block ciphers using mixed inner-and outer-round pipelining. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 94–102, 2001.

[7] Mathijs Cuppens. Implementation and Analysis of a ChaCha Hardware Accelerator for the Caddy Secure Webserver.

[8] Adam J Elbirt, Wei Yip, Brendon Chetwynd, and Christof Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):545–557, 2001.

[9] Reza Rezaeian Farashahi, Bahram Rashidi, and Sayed Masoud Sayedi. FPGA based fast and high-throughput 2-slow retiming 128-bit AES encryption algorithm. *Microelectronics journal*, 45(8):1014–1025, 2014.

[10] Umer Farooq and M Faisal Aslam. Comparative analysis of different AES implementation techniques for efficient resource usage and better performance of an FPGA. *Journal of King Saud University-Computer and Information Sciences*, 29(3):295–302, 2017.

[11] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded SystemsCHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings 3*, pages 251–261. Springer, 2001.

[12] Tim Good and Mohammed Benaissa. AES on FPGA from the fastest to the smallest. In *Cryptographic Hardware and Embedded Systems–CHES 2005: 7th International Workshop, Edinburgh, UK, August 29–September 1, 2005. Proceedings 7*, pages 427–440. Springer, 2005.

[13] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *Cryptology ePrint Archive*, 2016.

[14] Luca Henzen, Flavio Carbognani, Norbert Felber, and Wolfgang Fichtner. VLSI hardware evaluation of the stream ciphers Salsa20 and ChaCha, and the compression function Rumba. In *2008 2nd International Conference on Signals, Circuits and Systems*, pages 1–5. IEEE, 2008.

[15] Luca Henzen and Wolfgang Fichtner. FPGA parallel-pipelined AES-GCM core for 100G Ethernet applications. In *2010 Proceedings of ESSCIRC*, pages 202–205. IEEE, 2010.

[16] Van-Phuc Hoang, Van-Lan Dao, Cong-Kha Pham, et al. A compact, ultra-low power AES-CCM IP core for wireless body area networks. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–4. IEEE, 2016.

[17] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers 12*, pages 219–235. Springer, 2014.

[18] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in CryptologyCRYPTO99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*, pages 388–397. Springer, 1999.

[19] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in CryptologyCRYPTO96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[20] SV Dilip Kumar, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Low-Cost First-Order Secure Boolean Masking in Glitchy Hardware. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–2. IEEE, 2023.

[21] Thanikodi Manoj Kumar, Kasarla Satish Reddy, Stefano Rinaldi, Bidare Divakarachari Parameshachari, and Kavitha Arunachalam. A low area high speed FPGA implementation of AES architecture for cryptography application. *Electronics*, 10(16):2023, 2021.

[22] Qiang Liu, Zhenyu Xu, and Ye Yuan. High throughput and secure advanced encryption standard on field programmable gate array with fine pipelining and enhanced key expansion. *IET Computers & Digital Techniques*, 9(3):175–184, 2015.

[23] David McGrew. An interface and algorithms for authenticated encryption. Technical report, 2008.

[24] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. Technical report, 2018.

[25] Johannes Pfau, Maximilian Reuter, Tanja Harbaum, Klaus Hofmann, and Jürgen Becker. A hardware perspective on the ChaCha ciphers: Scalable Chacha8/12/20 implementations ranging from 476 slices to bitrates of 175 Gbit/s. In *2019 32nd IEEE International System-on-Chip Conference (SOCC)*, pages 294–299. IEEE, 2019.

[26] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 142–159. Springer, 2013.

[27] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Advances in Cryptology–CRYPTO 2015: 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I 35*, pages 764–783. Springer, 2015.

[28] Mikel Rodríguez, Armando Astarloa, Jesús Lázaro, Unai Bidarte, and Jaime Jiménez. System-on-Programmable-Chip AES-GCM implementation for wire-speed cryptography for SAS. In *2018 Conference on Design of Circuits and Integrated Systems (DCIS)*, pages 1–6. IEEE, 2018.

[29] Ronaldo Serrano, Ckristian Duran, Marco Sarmiento, Cong-Kha Pham, and Trong-Thuc Hoang. ChaCha20–Poly1305 Authenticated Encryption with Additional Data for Transport Layer Security 1.3. *Cryptography*, 6(2):30, 2022.

[30] Abolfazl Soltani and Saeed Sharifian. An ultra-high throughput and fully pipelined implementation of AES algorithm on FPGA. *Microprocessors and Microsystems*, 39(7):480–493, 2015.

[31] François-Xavier Standaert. Introduction to side-channel attacks. *Secure integrated circuits and systems*, pages 27–42, 2010.

[32] Joachim Strömbergson. Verilog 2001 implementation of the Chacha stream cipher. Available at `https://github.com/secworks/chacha` (25/04/2024).

[33] Eric J Swankoski, Richard R Brooks, Vijaykrishnan Narayanan, Mahmut Kandemir, and Mary Jane Irwin. A parallel architecture for secure FPGA symmetric encryption. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 132. IEEE, 2004.

[34] Xiphera. Chacha20-poly1305 product brief. Available at `https://xiphera.com/productbrief/ChaCha20Poly1305MPSoC.pdf`.

[35] Wei Yang and Anni Jia. Side-channel leakage detection with one-way analysis of variance. *Security and Communication Networks*, 2021:1–13, 2021.

[36] S-M Yoo, Deen Kotturi, DW Pan, and John Blizzard. An AES crypto chip using a high-speed parallel pipelined architecture. *Microprocessors and Microsystems*, 29(7):317–326, 2005.

[37] Joseph Zambreno, David Nguyen, and Alok Choudhary. Exploring area/delay tradeoffs in an AES FPGA implementation. In *International Conference on Field Programmable Logic and Applications*, pages 575–585. Springer, 2004.

[38] Xiwei Zhang, Meng Li, and Jing Hu. Optimization and implementation of AES algorithm based on FPGA. In *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, pages 2704–2709. IEEE, 2018.

[39] Yulin Zhang and Xinggang Wang. Pipelined implementation of AES encryption based on FPGA. In *2010 IEEE International Conference on Information Theory and Information Security*, pages 170–173. IEEE, 2010.

[40] Harshali Zodpe and Ashok Sapkal. An efficient AES implementation using FPGA with enhanced security features. *Journal of King Saud University-Engineering Sciences*, 32(2):115–122, 2020.

Abstract in lingua italiana

ChaCha20 è un cifrario a flusso simmetrico che prevede operazioni veloci e a basso costo rispetto ad AES. In questo lavoro proponiamo un'implementazione hardware del cifrario basata su unità quarter-round completamente combinatorie, quindi esploriamo tre diverse ottimizzazioni architetturali: dimensioni ridotte dei componenti, loop unrolling e pipelining. Esploriamo un design del cifrario protetto da attacchi side-channel che sfrutta la tecnica di implementazione a soglia, proponendo schemi di mascheramento per i suoi componenti aritmetici, quindi lo confrontiamo con una contromisura esistente di mascheramento a livello di porta logica. Riportiamo le cifre di merito di queste implementazioni su una FPGA Artix-7, ottenendo $5,62$ Gb/s con un design pipelined e $1,79$ Mb/s/slice con un design combinatorio. Valutiamo la sicurezza dei design protetti mediante analisi statistica della potenza.

Parole chiave: ChaCha20, cifrario, side-channel, masking, FPGA