A Practical Fault Attack on ARX-like Ciphers with a Case Study on ChaCha20

S V Dilip Kumar*, Sikhar Patranabis*, Jakub Breier†, Debdeep Mukhopadhyay*, Shivam Bhasin†, Anupam Chattopadhyay‡, and Anubhab Baksi‡

* *SEAL, Department of Computer Science and Engineering, IIT Kharagpur, India*
Email : {dilipkumar,sikhar.patranabis}@iitkgp.ac.in, debdeep@cse.iitkgp.ernet.in
† *Temasek Laboratories at Nanyang Technological University, Singapore*
Email : {jbreier,sbhasin}@ntu.edu.sg
‡ *School of Computer Science and Engineering, Nanyang Technological University, Singapore*
Email : anupam@ntu.edu.sg, anubhab001@e.ntu.edu.sg

***Abstract*—This paper presents the first practical fault attack on the ChaCha family of addition-rotation-XOR (ARX)-based stream ciphers. ChaCha has recently been deployed for speeding up and strengthening HTTPS connections for Google Chrome on Android devices. In this paper, we propose differential fault analysis attacks on ChaCha without resorting to nonce misuse. We use the instruction skip and instruction replacement fault models, which are popularly mounted on microcontroller-based cryptographic implementations. We corroborate the attack propositions via practical fault injection experiments using a laser-based setup targeting an Atmel AVR 8-bit microcontroller-based implementation of ChaCha. Each of the proposed attacks can be repeated with 100% accuracy in our fault injection setup, and can recover the entire 256 bit secret key using 5-8 fault injections on an average.**

***Keywords*-ChaCha, ARX cipher, Laser, Fault Attack, Instruction Skip, Instruction Replacement**

I. Introduction

The introduction of the Salsa family of stream ciphers by Bernstein in [1] has caused the Addition-Rotation-XOR (ARX) family of crypto-primitives to gain popularity in cryptographic literature. Besides Salsa, there exists today a number of block ciphers (e.g. SPECK [2]) and hash functions (e.g. BLAKE [3]) that use the ARX design paradigm. As the name suggests, ARX uses a combination of modular addition (for non-linearity), rotation and XOR operations, as opposed to the standard use of substitution-boxes (S-Boxes) along with linear operations. An ARX operation is typically of the form $d = ((a+c) \lll k) \oplus b)$ where k is usually a constant, and a, b, c and d are registers of appropriate width. ARX operations usually afford lower diffusion per round; consequently ARX-based crypto-primitives usually require a larger number of rounds. However, ARX-based designs avoid the need for S-Box lookups, which makes them efficient in software and resistant to timing-based side-channel attacks.

A. The ChaCha Family of Stream Ciphers

The ChaCha family of stream ciphers was proposed by Bernstein in [4]. It is an improvement over the original Salsa family of stream ciphers, with increased levels of diffusion per round. The basic operational unit for ChaCha is 32-bit words. The *ChaCha Function* maps a 256-bit key $k = \{k_0, k_1, ...k_7\}$, a 64-bit nonce $v = (v_0, v_1)$, and a 64-bit counter $t = (t_0, t_1)$ to a 512-bit keystream block. In particular, this function takes as input the 4×4 matrix of 32-bit words written as:

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & v_0 & v_1 \end{pmatrix} \quad (1)$$

where c_0, c_1, c_2 and c_3 are the predefined constants {0x61707865, 0x3320646E, 0x79622D32, 0x6B206574} respectively, and outputs a 512-bit keystream Z. The overall ChaCha function comprises of 20 rounds, where each round function is based on the following nonlinear operation (also called the *quarterround* function), which transforms a vector (x_0, x_1, x_2, x_3) to (y_0, y_1, y_2, y_3) by sequentially computing:

$$\begin{aligned} b_0 = x_0 + x_1, b_3 = (x_3 \oplus b_0) \lll 16 \\ b_2 = x_2 + b_3, b_1 = (x_1 \oplus b_2) \lll 12 \\ y_0 = b_0 + b_1, y_3 = (b_3 \oplus y_0) \lll 8 \\ y_2 = b_2 + y_3, y_1 = (b_1 \oplus y_2) \lll 7 \end{aligned} \quad (2)$$

The quarterround function is applied to the state matrix in row-major and diagonal-major fashion in the even and odd numbered rows, respectively. Let X^r be the state matrix after round r. The final output keystream block Z is computed as $Z = X \boxplus X^{20}$, where $\boxplus$ denotes word-wise integer addition.

B. Our Contributions

In this paper, we propose four differential fault analysis (DFA) attacks on ChaCha using the instruction skip and

Table I: The ChaCha quarterround Function.

Round	ARX Input Vectors to the Round Function
Odd numbered	(x_0,x_5,x_{10},x_{15}), (x_1,x_6,x_{11},x_{12}), (x_2,x_7,x_8,x_{13}), (x_3,x_4,x_9,x_{14})
Even numbered	(x_0,x_1,x_2,x_3), (x_4,x_5,x_6,x_7), (x_8,x_9,x_{10},x_{11}), $(x_{12},x_{13},x_{14},x_{15})$

instruction replacement fault models [5] that usually target microcontroller-based cryptographic implementations. Our attacks target the keystream generation module at the decryption site, and entirely avoid nonce misuse. We practically demonstrate our proposed attacks on an Atmel AVR 8-bit microcontroller-based implementation of ChaCha using a laser fault injection setup. Each of our proposed attacks using instruction skips requires around **5-8 fault injections** on an average to recover the entire 256 bit secret key, while the attack using instruction replacements requires **32 fault injections** on an average for full key recovery. To the best of our knowledge, this is the first practical demonstration of fault attacks on the ChaCha family of stream ciphers to be reported in the literature.

II. Differential Fault Analysis of Stream Ciphers: Avoiding Nonce-Misuse

Standard stream cipher encryption and decryption are usually accompanied by the use of a nonce and a counter which do not repeat. This could lead to a notion that classical DFA [6], [7] using a correct and faulty pair of encryption executions on the same plaintext may not apply in case of stream ciphers. In this section, we point to a simple attack model where the adversary can mount classical DFA on stream ciphers by targeting the decryption module instead.

Our proposed attack model is illustrated in Fig. 1. The attack steps may be enumerated as follows:

1) The adversary is assumed to *know* both the plaintext message M and the ciphertext C during encryption. This obviously implies that the adversary also knows the correct keystream K.
2) The target for fault injection is the equivalent keystream generation algorithm during decryption. The adversary obtains the faulty message M' and recovers the faulty keystream $K' = M' \oplus C$.
3) The adversary now proceeds with the standard DFA procedure using the knowledge of K and K'.

Since the nonce and counter values are identical during both encryption and decryption, nonce misuse is completely avoided. Thus, as long as the adversary can obtain at least a part of the secret key from the correct and faulty pair of output keystreams, the whole key can always be recovered from multiple fault injections targeting different parts of the key. In summary, *classical fault analysis techniques such as DFA can be practically mounted on stream ciphers without misusing the nonce in any way*.

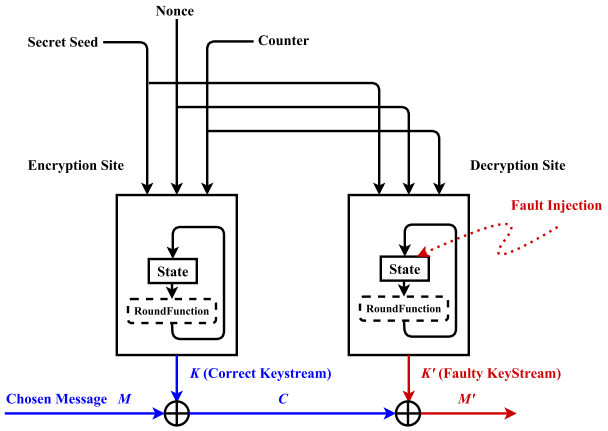

Figure 1: DFA on Stream Ciphers: Attack Model without Nonce Misuse.

III. Instruction Skips and Instruction Replacement Attacks

In this paper, we present four instances of DFA on ChaCha under the attack model described in Section II that avoids nonce misuse entirely. We choose the *assembly instruction replacement fault model*, which has been widely studied in the literature [5] as a potent threat to microcontroller-based cryptographic implementations. A widely studied sub-class of instruction replacement attacks is the instruction *skip* model, where an instruction is essentially replaced by a NOP. In our attacks, we also consider the more general class of instruction replacement attacks, where the target instruction is converted to an alternate instruction acting on the same operands. We target the following operations of ChaCha in the different attacks proposed in this paper:

- The final addition operation between the input, X, and the output of round 20, X^{20}, during the keystream generation algorithm
- The 12-bit rotation operation during a quarterround in round 20 of the keystream generation algorithm
- A branch-not-equal operation in the last quarterround in round 20 of the keystream generation algorithm

We point out here that our attack techniques are generic and not specifically dependent on the target implementation chosen for this paper. While the exact fault injection parameters may be adopted to the target architecture, the basic principle of the attacks would remain the same. In particular, any unprotected microcontroller-based implementation of ChaCha is expected to contain similar instructions

Table II: Inputs to the quarterround Function.

Content of $r19$	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
Input Vector Indices to quarterround	(0,4,8,12)	(1,5,9,13)	(2,6,10,14)	(3,7,11,15)	(0,5,10,15)	(1,6,11,12)	(2,7,8,13)	(3,4,9,14)

corresponding to the operations that we target in our attack.

IV. The Target Implementation of ChaCha

We target an Atmel AVR microcontroller-based implementation of ChaCha. The ISA for this microcontroller comprises of standard instructions such as `add`, `sub`, `mov` and `cp`. The microcontroller uses a 16-bit address-space, while being an 8-bit processor. Consequently, most of the instructions use 8-bit operands. Our implementation uses the general-purpose register set $(r16, r17, \cdots, r25)$, and the additional register set $(r26, r27, \cdots, r31)$ for indirect addressing.

A. Implementation of the quarterround Function

We present snippets of the AVR code for implementing the quarterround function in Listing 1. The register $r19$ stores the input vector for the quarterround operation. An example comparison step with $r19$ is illustrated in Line 2. If $r19$ stores 0x00, then the first column comprising of x_0, x_4, x_8, x_{12} is treated as the input, and the instructions between Line 4 and Label 80 are executed. On the other hand, if $r19$ does not hold 0x00, the control branches to Label 40, followed by another immediate branch to Label 80. The process repeats iteratively until a matching condition for $r19$ is encountered. The use of two conditional branch instructions is motivated by the limitation on the range of addresses that a single jump can perform in the Atmel ISA. The input vector indices to the quarterround function for different values of the register $r19$ are depicted in Table II.

The fault attacks proposed in this paper target one or more instructions using a laser fault injection setup, as described next. The aim of injection is to skip/alter the target instructions, resulting in alterations to the control flow of the original program, which are subsequently exploited to efficiently retrieve the key.

Listing 1: Subroutine *ARX* : Implementation of quarterround function.

```
ARX%=:
cpi r19, 0x00
brne 40f
/* Column 1 : (0,4,8,12)*/
ldi r27,hi8(Array)
ldi r26,lo8(Array)
ldi r29,hi8(State + 0x00)
ldi r28,lo8(State + 0x00)
ldi r31,hi8(State + 0x10)
ldi r30,lo8(State + 0x10)
/*Relative call to Plus32*/
rcall Plus32%=
ldi r29,hi8(State + 0x30)
ldi r28,lo8(State + 0x30)
ldi r31,hi8(State + 0x00)
ldi r30,lo8(State + 0x00)
/*Relative call to Xor32*/
rcall Xor32%=
ldi r29,hi8(State + 0x30)
ldi r28,lo8(State + 0x30)
ldi r31,hi8(State + 0x30)
ldi r30,lo8(State + 0x30)
/*Relative call to Rotate32*/
ldi r22, 0x10
rcall Rotate32%=
...
...
40:
cpi r19, 0x00
brne 80f
...
...
80:
cpi r19, 0x01
brne 41f
...
...
/*Rest of the code*/
```

V. Laser-Based Experimental Setup

In this section we will explain the experimental setup which was used for the fault injection.

We describe the various components of the fault injection setup below:

- **Device under test.** As the device under test (DUT), we have selected a general-purpose 8-bit microcontroller, Atmel ATmega328P. It operates at 16 MHz, therefore one clock cycle takes 62.5 ns. The area of the chip is 3×3 mm^2 large. Before the experiment, the DUT was decapsulated by mechanical milling tools and polished by using Ultra-TEC ASAP-1 sample preparation equipment, in order to provide enough precision for the laser beam. This chip was mounted on the Arduino UNO board, adjusted for the laser testing purposes. The board communicates with the PC using the USBCDC interface.
  We set a trigger signal on the board to HIGH (5 V) before performing the operations to correctly identify the desired time. The board was mounted on an a positioning table with the step precision of 0.05 μm.
- **Optical source.** We utilized a near-infrared diode pulse laser (1064 nm) with maximal pulse power 20 W. The power was further reduced to 8 W by using a $20\times$ magnifying lens. Laser spot size with this lens is 15×3.5 μm^2 and response to trigger pulse is $\approx$ 100 ns.

A trigger device was used for capturing the trigger signal from the device and adjusting the laser activation timing. A digital sampling oscilloscope was used for verifying the laser pulse activation time and for estimating the timing of sub-operations in the cipher implementation. Finally, a PC workstation was used for communication with the DUT and analysis of results.

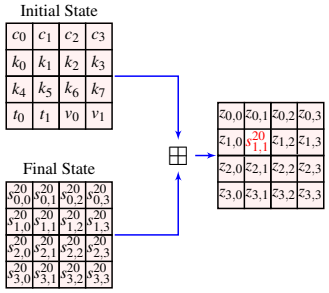

Figure 2: First Instruction Skip Attack: Recovering k_1.

VI. Instruction Replacement-Based DFA of ChaCha: Detailed Description

In this section, we describe four DFA attacks on ChaCha using the instruction replacement faults injected with the help of laser fault injection setup described above. The faults target the implementation of the ARX quarterround, described in Listing 1, as well as the final addition between the initial state X and the output, X^{20} of round 20 of the keystream generation algorithm.

A. Attack on Final Addition

1) Attack Description: The first DFA attack targets the final word-wise addition of the initial state containing the constants, key, counter and nonce to the final state generated after round 20. The attack skips one of the 16 word-additions, causing the faulty keystream output to trivially reveal the corresponding word of the secret key. The DFA procedure for recovering the key word k_1 is depicted in Fig. 2. Let Z and $Z^{'}$ be the correct and faulty keystreams respectively. Then, we have:

$$\begin{aligned} z_{1,1} &= k_1 + s^{20}_{1,1} \\ z^{'}_{1,1} &= s^{20}_{1,1} \\ k_1 &= z_{1,1} - z^{'}_{1,1} \end{aligned} \tag{3}$$

Solving the set of Equations in (3), we recover the key word k_1. Note that subtraction is performed *modulo* 2^{32}. The attack may now be repeated on each of the 8 words in the second and third row of the keystream, to retrieve each of the 8 words of the secret key.

2) Attack Realization: We now discuss a practical fault injection setting to mount the above attack on our AVR-based implementation of ChaCha. In the implementation, the 16-bit addresses of the words to be added are loaded in indirect addressing registers, and a 32-bit addition subroutine is invoked. In order to inject the desired fault in our laser-based setup, we set the trigger to just before the commencement of the final addition operation. The injection successfully skipped the function call to the 32-bit addition subroutine, and trivially revealed the corresponding word in the secret key. The fault injection was found to be 100% repeatable for each of the 8 potential target words, with varying injection timings. Detailed experimental results for the attack are presented in Section VII-A.

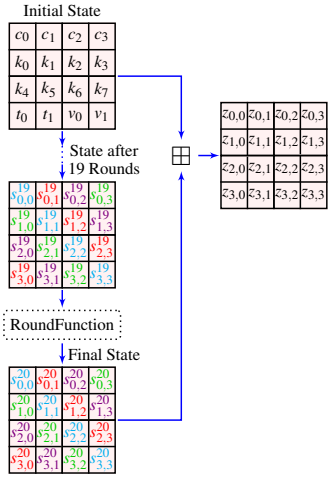

Figure 3: Second Instruction Skip Attack: Perturbing Rotation Offset.

B. Attack on Rotation

In any ARX-based cipher, the intermediate state is circular-rotated by a certain number of bits at some stage during quarterround execution. In ChaCha, for example, the various intermediate state registers are rotated by an offset of 16, 12, 8 and 7 bits in each quarterround. In our second attack, we target one such rotation in each quarterround of round 20 in the key generation algorithm. We inject faults to perturb the rotation offset, and exploit the resulting keystream via a DFA to recover two words of the secret key.

1) Attack Description: We describe a fault attack that targets the 12-bit circular rotation in each quarterround of round 20 in the key generation algorithm. The attack essentially skips a single-bit circular shift, resulting in a 11-bit overall rotation instead of the expected 12-bit rotation. Now, let Z and $Z^{'}$ be the correct and faulty keystreams, respectively. Equations (2) and (4) describe the quarterround operations for the correct and faulty executions of the quarterround, respectively.

$$\begin{aligned} b_0 = x_0 + x_1, b_3 = (x_3 \oplus b_0) \lll 16 \\ b_2 = x_2 + b_3, b^{'}_1 = (x_1 \oplus b_2) \lll 11 \\ y^{'}_0 = b_0 + b^{'}_1, y^{'}_3 = (b_3 \oplus y^{'}_0) \lll 8 \\ y^{'}_2 = b_2 + y^{'}_3, y^{'}_1 = (b^{'}_1 \oplus y^{'}_2) \lll 7 \end{aligned} \tag{4}$$

The inputs x_0, x_1, x_2, x_3 in Equations (2) and (4) are the main diagonal words of the state after round 19, (see $\{s^{19}_{0,0}, s^{19}_{1,1}, s^{19}_{2,2}, s^{19}_{3,3}\}$ in Fig. 3). Similarly, the outputs $\{y_0, y_1, y_2, y_3\}$ and $\{y^{'}_0, y^{'}_1, y^{'}_2, y^{'}_3\}$ are the main diagonal words after *Round* 20, $\{s^{20}_{0,0}, s^{20}_{1,1}, s^{20}_{2,2}, s^{20}_{3,3}\}$. Our aim is to recover the final state words in the first diagonal, namely

$\{y_0, y_1, y_2, y_3\}$, which then trivially reveal the corresponding words in the secret key.

Observe that after the completion of round 20, the first and fourth rows of final state are added to constants, counter values and nonce values, all of which are publicly available. This in turn reveals the following correct and faulty outputs of the quarterround: $\{y_0, y_3, y_0^{'}, y_3^{'}\}$ (see the system of equations below):

$$\begin{aligned} y_0 &= z_{0,0} - c_0, y_0^{'} = z_{0,0}^{'} - c_0 \\ y_3 &= z_{3,3} - v_1, y_3^{'} = z_{3,3}^{'} - v_1 \end{aligned} \tag{5}$$

Now, the difference between the correct and faulty keystreams reveal their difference in the respective final states after round 20, leading to the computation of both b_0 and b_1 as:

$$b_1 = b_1^{'} \lll 1$$

that is,

$$b_1^{'} = \begin{cases} (b_1 + 2^{32} - 1)/2 & \text{if LSB}(b_1) = 1 \\ b_1/2 & \text{otherwise} \end{cases}$$

$$\begin{aligned} b_1 - b_1^{'} &= z_{0,0} - z_{0,0}^{'} = y_0 - y_0^{'} \\ y_2 - y_2^{'} &= z_{2,2} - z_{2,2}^{'} = y_3 - y_3^{'} \\ y_1 - y_1^{'} &= z_{1,1} - z_{1,1}^{'} \end{aligned} \tag{6}$$

Substituting the values we computed from the set of Equations (6), we just have y_2 left to solve for finding the key words, k_1 and k_6. Now, observe the following relation:

$$y_1 - y_1^{'} = ((b_1 \oplus y_2) \lll 7) - ((b_1^{'} \oplus (y_2 - z_{2,2} + z_{2,2}^{'}) \lll 7)$$

where y_2 is the only unknown. A simulation over all 2^{32} values of y_2 reveals that on an average, the above equation is satisfied by 2^{10} values of y_2, which in turn maps to 2^{10} unique pairs of values for (k_1, k_6) (see the following Equations (7)).

$$\begin{aligned} y_1^{'} &= (b_1^{'} \oplus y_2^{'}) \lll 7 \\ y_1 &= (b_1 \oplus y_2) \lll 7 \\ k_1 &= z_{1,1} - y_1 = z_{1,1}^{'} - y_1^{'} \\ k_6 &= z_{2,2} - y_2 = z_{2,2}^{'} - y_2^{'} \end{aligned} \tag{7}$$

Thus the fault attack reduces the search space for the pair (k_1, k_6) from 2^{64} to 2^{10}. Repeating the same experiment for the three other diagonals finally reduces the search space of the entire 256-bit secret key from 2^{256} to around 2^{12}.

2) Attack Realization: In the AVR-based implementation of the quarterround, the attack on the rotation of the state registers is realized as follows:

- When the number of bits to be rotated is a multiple of 8, the circular rotation is a simple permutation of the 8-bit registers. In this case, there is no single-bit shift operation that could be skipped using the laser injection setup.
- However, when the number of bits to be rotated is *not* a multiple of 8, say 12, then the register permutation must be followed by four single-bit circular shifts, performed by four relative calls to single-bit circular shift subroutine, one of which is targeted with 100% repeatability using the laser injection setup.

```
    ARX%=:
    cpi r19, 0x00
    brne 40f
    ...
5   40:                 Column 1 :(0,4,8,12)
    cpi r19, 0x00
    brne 80f
    ...
    80:
10  cpi r19,0x01
    ...
    83:
    cpi r19, 0x04
    brne 44f
15  ...
    44:                 Diagonal 1 :(0,5,10,15)
    cpi r19, 0x04
    brne 84f
    ...
20  84:
    cpi r19, 0x05
    brne 45f
    ...
    45:                 Diagonal 2 :(1,6,11,12)
25  cpi r19, 0x05
    brne 85f
    ...
    85:
    cpi r19, 0x06
30  brne 46f
    ...
    46:                 Diagonal 3 :(2,7,8,13)
    cpi r19, 0x06
    brne 86f
35  ...
    86:
    cpi r19, 0x07
    brne 47f
    ...
40  47:                 Diagonal 4 :(3,4,9,14)
    cpi r19, 0x07
    brne 87f
    ...
    87:
45  rjmp ARXExit%=
```

Figure 4: Branching in ChaCha quarterround function.

In particular, the laser trigger may be set just before round 20, with the $r19$ register holding the main diagonal entries to be passed to the quarterround as inputs. Detailed experimental results for the attack are presented in Section VII-B.

C. A Diagonal Fault Attack via Alteration of Control Flow

Our third attack is a demonstration of a diagonal fault attack via a change in the control flow of the target program in round 20 of the keystream generation algorithm. Recall that Listing 1 depicted the use of branching instructions in the AVR code for the quarterround operation of ChaCha. In particular, observe in Fig. 4 that the operation on any diagonal involves a series of `cpi-brne` instructions, before the program control arrives at a matching label for the register $r19$.

Unlike in the previous fault injection instances, where the trigger was placed before round 20, in this case, we place it right before the ARX call for diagonal 4 in round 20 (that is, after the executions for diagonals 1, 2 and 3 are completed). Once the subroutine ARX is invoked, the control spends the initial clock cycles in executing compare and jump instructions before it starts the operations for diagonal 4. It was observed that one of the `brne` instructions got skipped, thereby preventing the control from branching out to the next label. In particular, a `brne` instruction skip was observed following the labels 83, 84 and 85 in Fig. 4 (Line numbers 14, 22, and 30). *Targeting line 14 caused the instructions between labels 83 and 44 to be executed twice instead of once*; for correct functionality, *this part of the code should have been skipped on the second occasion*. A similar pattern is detected for the `brne` instructions corresponding to the other diagonals too. Recall that the reason for using two jumps per diagonal is because of the limitation on the address range that a `brne` instruction could incorporate. More specifically, a jump from label 83 to 84 would not be possible with a single branch. The section of the code between labels 83 and 44 perform the *Round* operation on Diagonal 1, except the 7-bit rotation which happens after label 44. The operations that would take place are shown in the equations below.

$$\begin{aligned} b_0 &= x_0 + x_1, b_3 = (x_3 \oplus b_0) \lll 16 \\ b_2 &= x_2 + b_3, b_1 = (x_1 \oplus b_2) \lll 12 \\ y_0 &= b_0 + b_1, y_3 = (b_3 \oplus y_0) \lll 8 \\ y_2 &= b_2 + y_3, y_1 = (b_1 \oplus y_2) \end{aligned} \tag{8}$$

Further, observe that the inputs (x_0, x_1, x_2, x_3) are nothing but $(s^{20}_{0,0}, s^{20}_{1,1}, s^{20}_{2,2}, s^{20}_{3,3})$, since the fault takes place only the last diagonal undergoes the quarterround operation. Next, we solve the following equations to retrieve the key words k_1 and k_6.

$$\begin{aligned} x_0 = z_{0,0} - c_0, x_3 = z_{3,3} - v_1 \\ y_0 = z^{'}_{0,0} - c_0, y_3 = z^{'}_{3,3} - v_1 \end{aligned} \tag{9}$$

$$\begin{aligned} b_3 &= (y_3 \ggg 8) \oplus y_0 \\ b_0 &= (b_3 \ggg 16) \oplus x_3 \\ x_1 &= b_0 - x_0 \end{aligned} \tag{10}$$

$$\begin{aligned} b_1 &= y_0 - b_0 \\ b_2 &= (b_1 \ggg 12) \oplus x_1 \\ x_2 &= b_2 - b_3 \end{aligned} \tag{11}$$

$$\begin{aligned} y_2 &= b_2 + y_3 \\ y_1 &= b_1 \oplus y_2 \\ k_1 &= z^{'}_{1,1} - y_1 = z_{1,1} - x_1 \\ k_6 &= z^{'}_{2,2} - y_2 = z_{2,2} - x_2 \end{aligned} \tag{12}$$

As shown in Equation (9), the values of x_0 and x_3 are revealed from the non-faulty keystream. This information is enough to obtain all other $x_i s$, $b_i s$ and $y_i s$ of the faulty stream, thus reducing the search space of (k_1 and k_6) to a unique pair of values. Following an identical fault injection approach, the values of k_2, k_3, k_4, k_7 can also be retrieved. *Thus the overall key search space is reduced from* 2^{256} *to* 2^{64} *using three fault injections*. The remaining key words k_0 and k_5 may be further recovered using two additional instruction skips. We have already discussed in Section VI-A an instruction skip attack on the final addition step in the keystream generation algorithm. Since every quarterround of the keystream generation algorithm also has four 32-bit addition operations, skipping the first and third addition operations in the last quarterround (corresponding to diagonal 4) of round 20 allows us to uniquely retrieve k_0 and k_5 from the following set of equations, respectively:

$$\begin{aligned} b^{''}_0 = x_0, b^{''}_3 = (x_3 \oplus b^{''}_0) \lll 16 \\ b^{''}_2 = x_2 + b^{''}_3, b^{''}_1 = (x_1 \oplus b^{''}_2) \lll 12 \\ y^{''}_0 = b^{''}_0 + b^{''}_1, y^{''}_3 = (b^{''}_3 \oplus y^{''}_0) \lll 8 \\ y^{''}_2 = b^{''}_2 + y^{''}_3, y^{''}_1 = (b^{''}_1 \oplus y^{''}_2) \lll 7 \end{aligned} \tag{13}$$

$$\begin{aligned} b^{''}_3 &= (y^{''}_3 \ggg 8) \oplus y^{''}_0 \\ x_0 &= b^{''}_0 = (b^{''}_3 \ggg 16) \oplus x_3 \\ x_1 &= b_0 - x_0 \\ b_2 &= (b_1 \ggg 12) \oplus x_1 \\ y_2 &= b_2 + y_3 \\ y_1 &= (b_1 \oplus y_2) \lll 7 \end{aligned} \tag{14}$$

$$\begin{aligned} b_0 = x_0 + x_1, b_3 = (x_3 \oplus b_0) \lll 16 \\ b_2 = x_2 + b_3, b_1 = (x_1 \oplus b_2) \lll 12 \\ y^{'}_0 = b_0, y^{'}_3 = (b_3 \oplus y^{'}_0) \lll 8 \\ y^{'}_2 = b_2 + y^{'}_3, y^{'}_1 = (b_1 \oplus y^{'}_2) \lll 7 \end{aligned} \tag{15}$$

$$\begin{aligned} y_0 &= z_{0,0} - c_0 \\ y^{'}_0 &= z^{'}_{0,0} - c_0 \\ y_3 &= z_{3,3} - v_1 \\ b_1 &= y_0 - y^{'}_0 \\ b_0 &= y_0 - b_1 \\ b_3 &= (y_3 \ggg 8) \oplus y_0 \\ x_3 &= (b_3 \ggg 16) \oplus b_0 \end{aligned} \tag{16}$$

$$\begin{aligned} k_0 = z_{1,1} - y_1 \\ k_5 = z_{2,2} - y_2 \end{aligned}$$

By skipping the third addition the values of b_1 and b_0 can be found by solving the Equations in (15) and (16) , and similarly x_0, x_1 can be derived by skipping the first addition, refer to Equations (13) and (14). Every other unknown can be found by substituting these values, including the key words k_0 and k_5. Hence we have uniquely identified the

256-bit key using 5 distinct faults injected targeting different instructions. Detailed experimental results for this attack are presented in Section VII-C.

D. Instruction Replacement Attack on the Final Addition

Contrary to the previous attacks that exploited the instruction skip fault model, our final attack exploits an instruction *replacement* fault, where the intended instruction is replaced by an alternate instruction with the same set of operands.Similar to the attack described in Section VI-A, this attack also targets the final addition operation between the initial and final states of the keystream generation algorithm. However, instead of skipping word-wise additions, the attack replaces the `add` instruction by a `sub` instruction and `adc` instruction by a `sbc` instruction (the suffix `c` denotes carry-based operations). Note that unlike previous attacks, this attack is platform dependent and the instruction conversion has been observed for the DUT chosen for this paper. The attack procedure is described next.

Observe that while in the first attack, we directly skipped the whole 32-bit addition operation, in this attack, we inject fault at finer granularity on the four 8-bit addition operations within the 32-bit module. Let $s = (s_{31}s_{30}...s_0)$ be one of the words in the final state X^{20} of the keystream generation algorithm, which is added with a corresponding word $k = (k_{31}k_{30}...k_0)$ of the secret key to obtain a word $z = (z_{31}z_{30}...z_0)$ in the output keystream Z. Let $z^1 = (z^1_{31}z^1_{30}...z^1_0)$ be the faulty keystream generated upon skipping the first addition operation, $z^2 = (z^2_{31}z^2_{30}...z^2_0)$ be the faulty keystream generated upon skipping the second addition operation, and so on for z^3 and z^4. The first of these additions typically does not involve a carry as it adds the lowest 8 bits, while the remaining additions involve distinct carry elements.

The following relations are straightforward to observe upon fault injection in each of the four additions:

$$\begin{aligned} z_7z_6...z_0 &= s_7s_6...s_0 + k_7k_6...k_0 \\ z_{15}z_{14}...z_8 &= s_{15}s_{14}...s_8 + k_{15}k_{14}...k_8 + c_0 \\ z_{23}z_{22}...z_{16} &= s_{23}s_{22}...s_{16} + k_{23}k_{22}...k_{16} + c_1 \\ z_{31}z_{30}...z_{24} &= s_{31}s_{30}...s_{24} + k_{31}k_{30}...k_{24} + c_2 \end{aligned} \quad (17)$$

$$\begin{aligned} z^1_7z^1_6...z^1_0 &= s_7s_6...s_0 - k_7k_6...k_0 \\ z^1_{15}z^1_{14}...z^1_8 &= s_{15}s_{14}...s_8 + k_{15}k_{14}...k_8 + c^1_0 \\ z^1_{23}z^1_{22}...z^1_{16} &= s_{23}s_{22}...s_{16} + k_{23}k_{22}...k_{16} + c^1_1 \\ z^1_{31}z^1_{30}...z^1_{24} &= s_{31}s_{30}...s_{24} + k_{31}k_{30}...k_{24} + c^1_2 \end{aligned} \quad (18)$$

Solving equations (17) and (18) together reveals the final state byte $s_7s_6...s_0$, the secret key byte $k_7k_6...k_0$ and the carry c_0. Similarly, comparing z^1 and z^2 would expose the next byte of the final state and the secret key, and so on. The same fault injection method can be followed for the other key words as well, and the whole secret key can thus be recovered with a total of 32 fault injections on an average. Detailed experimental results for this attack are presented in Section VII-D.

Table III: Fault Injection Timings for Final Addition Skip & Single-bit Rotation.

Final Addition Skip		Single-bit Rotation	
Key Words Revealed	Fault Injection Timing (in clock cycles)	Key Words Revealed	Fault Injection Timing (in clock cycles)
k_0	t_0 + 153	(k_1,k_6)	t_1 + 272
			t_1 + 301
k_1	t_0 + 191		t_1 + 330
			t_1 + 359
k_2	t_0 + 230	(k_2,k_7)	t_1 + 1087
			t_1 + 1116
k_3	t_0 + 281		t_1 + 1145
			t_1 + 1174
k_4	t_0 + 319	(k_3,k_4)	t_1 + 1906
			t_1 + 1935
k_5	t_0 + 359		t_1 + 1964
			t_1 + 1993
k_6	t_0 + 398	(k_0,k_5)	t_1 + 2729
			t_1 + 2758
k_7	t_0 + 436		t_1 + 2787
			t_1 + 2816

VII. Experimental Results

In this section, we present experimental results for the attacks described in Section VI using the laser setup described in Section V. The attacks target the keystream generation in the decryption site, as described in Section II.

A. Experimental Results: Skipping the Final Addition

This experiment corresponds to the attack described in the subsection VI-A. In this experiment, we set the laser trigger just before final addition commences(that is, right after round 20 ends). The word which gets skipped depends on the timing of the fault injection. The timings for which the 8 key words skip addition are listed in Table III. Let t_0 be the number of clock cycles elapsed from the start of encryption to the beginning of the Final Addition.

B. Experimental Results: Attack on Rotation

In this experiment we target the 4 relative calls to single-bit circular shift subroutine of 12-bit rotation. Table III shows four different fault injection timings which would skip one of the four *rcall*s. On average, repeating the experiment twice for each pair of key words would uniquely determine the key pair. Let t_1 be the number of clock cycles elapsed from the start of encryption to the beginning of round 20.

C. Experimental Results: Diagonal Fault Attack via Alteration of Control Flow

This experiment targets branch instructions in specific to be skipped, recall the attack in Section VI-C . Skipping three `brne` instructions uniquely reveal 6 words of key in total, the details about their timings and the words revealed are mentioned in Table IV. Note that t_2 is the time elapsed from the start of the encryption to the beginning of quarterround for the last diagonal ($\sim t_1$+2445).

Table IV: Fault Injection Timings for Diagonal Fault Attack.

Diagonal Repeated	Key Words Revealed	Fault Injection Timing (in clock cycles)
(x_0,x_5,x_{10},x_{15})	(k_1,k_6)	t_2 + 17
(x_1,x_6,x_{11},x_{12})	(k_2,k_7)	t_2 + 25
(x_2,x_7,x_8,x_{13})	(k_3,k_4)	t_2 + 32

Table V: Summary of Fault Attack Results on ChaCha.

Attack Type	Number of Fault Injections	Key Space
Attack on final Addition	8	1
Attack on Rotation	8	1
Diagonal Fault Attack	3	2^{64}
	5	1
Instruction Replacement	32	1

D. Experimental Results: Instruction Replacement Attack

In the final experiment, every fault injection leaks 8 bits of the secret key. For instance, to retrieve the key word k_0, we vary the fault injection timings between $t_0 + 153$ and $t_0 + 191$. Changing the timings within this range would cause four different `add` instructions to be skipped and eventually 32 bits of the key is revealed. Similar timing variations reveal other key words as well.

In summary, Table V shows the number of fault injections needed to obtain the 256-bit key using the aforementioned attacks. Furthermore, a powerful attacker can perform multiple fault injections in the same decryption cycle, to target different key bytes with a single faulty key stream. This would drastically reduce the number of faulty keystreams required to obtain the whole secret key, as all the words are treated independently.

VIII. Discussion

The fault models described in this paper were based on instruction skip/replacement attacks. However, the attacks would still hold in the presence of more traditional fault models, such as stuck-at and transient randomized faults, as described next.

- **Attack on the Final Addition** A single word *stuck-at-0* fault at the target key words in the initial state during final addition produces the same result as the instruction skip of relative call to the 32-bit addition subroutine.
- **Attack on Rotation** In this attack, a single word is directly affected in the quarterround function (e.g. b_1 in (4)). The attack could be equivalently performed by *injecting a randomized fault in the same keyword*.
- **Diagonal Fault Attack** The diagonal fault attack via alteration of control flow could be achieved by injecting *multiple byte stuck-at faults in the first and fourth row of the state matrix*. This would avoid the need for diagonal re-execution, since it directly reveals inputs x_0 and x_3 for the quarterround functions in round 20. The same attack procedure follows subsequently.
- **Instruction Replacement** This instruction replacement attack could alternatively be performed via injection of transient faults in the target byte, followed by the same differential analysis as presented in Section VI-D.

The attacks on the quarterround functions are generally preferred over the straightforward attacks on final addition as they involve lesser number of fault injections. For instance, we need 256 bit flips in the final addition to reveal the 256-bit key, but if we are able to flip the bits of a input word for the quarterround functions in the last round, we would require less than 128 bit flips to get the whole key owing to fault propagation by modular addition.

IX. Conclusion

In this paper, we presented three instances of instruction skip attacks and one instance of instruction replacement attack targeting, and also demonstrated them practically on an AVR implementation using a laser-based fault injection setup. Our attacks targeted the keystream generation module at the decryption site, and entirely avoided nonce misuse. We practically demonstrated our proposed attacks on an Atmel AVR microcontroller-based implementation of ChaCha using a laser fault injection setup. Each of our proposed attacks using instruction skips requires around 5-8 fault injections on an average to recover the entire 256 bit secret key, while the attack using instruction replacements requires 32 fault injections on an average for full key recovery. All the attacks proposed in the paper were repeatable in our laser-based FI setup with 100% accuracy. To the best of our knowledge, this was the first practical demonstration of fault attacks on the ChaCha family of ARX-based stream ciphers to be reported in the literature.

References

[1] D. J. Bernstein, "The salsa20 family of stream ciphers," in *New Stream Cipher Designs - The eSTREAM Finalists*, 2008, pp. 84–97.

[2] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK lightweight block ciphers," in *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, 2015, pp. 175:1–175:6.

[3] J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan, "Sha-3 proposal blake," *Submission to NIST*, 2008.

[4] D. J. Bernstein, "Chacha, a variant of salsa20," in *Workshop Record of SASC*, vol. 8, 2008.

[5] D. Naccache, "Finding faults [data security]," *IEEE Security & Privacy*, vol. 3, no. 5, pp. 61–65, 2005.

[6] J. Blömer and J.-P. Seifert, "Fault based cryptanalysis of the advanced encryption standard (aes)," in *Computer Aided Verification*. Springer, 2003, pp. 162–181.

[7] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *IFIP International Workshop on Information Security Theory and Practices*. Springer, 2011, pp. 224–233.