An Improved Differential Fault Attack on the Stream Cipher Espresso *

Debendranath Das[1][0000−0002−3855−1700], Anirban Ghatak[1][0000−0002−8654−2411], Indivar Gupta[2], Sabyasachi Karati[1], and Arindam Mandal[2]

[1] Indian Statistical Institute, 203, B. T. Road, Kolkata, India, 700108
[2] Scientific Analysis Group, DRDO, Metcalfe House, Delhi, India, 110054

Abstract. A recent paper by Bathe *et al.* proposed a differential fault injection attack on the stream cipher Espresso. The authors claimed that the injection of 4 random faults in the internal state was enough for state recovery. In this article we present an improved implementation of their technique - with modifications in both the pre-processing and the online phases. As a result we report internal state recovery of Espresso using only 3 injected faults in comparable time. Our experimental data also leads to some interesting observations regarding the dependence of the stages of the Espresso state.

Keywords: Differential fault attack · Espresso cipher.

1 Introduction

The stream cipher Espresso was proposed by E. Dubrova and M. Hell [1] with the design rationale of "minimizing the hardware footprint and maximizing the throughput of the design"(page 274, [1]), which would make it a candidate scheme for providing security in 5G mobile communication. The main design feature of Espresso is the use of a non-linear feedback shift register (NLFSR) in the so-called *Galois configuration*, allowing non-linear feedback from multiple shift register stages into several intermediate stages of the shift register. This is claimed to result in "smaller" (cf. Section 4, [1]) feedback functions leading to a reduction in the overall propagation delay. The Espresso NLFSR is thus different from the more widely reported FSRs in the so-called *Fibonacci configuration*, where the feedback function is input only at the terminal stage and the only input to each intermediate stage is from its preceding stage.
Since its proposal in 2017, the application of the following major cryptanalysis techniques has been reported on Espresso - differential fault attack [2], TMDTO attack [3], algebraic attack [5]. As the main focus of our paper is the differential fault attack, its is treated in detail in the subsequent sections. In [3], a possible TMDTO attack on Espresso is discussed along the lines of [4], where a

* The work of the first and second authors was supported by the project "Analysis of Cryptographic Algorithms and Protocols used in Wireless Mobile Communications" of SAG, DRDO.

pre-determined set of 39 internal state bits of Espresso are set to zero, and the update equations are solved to extract the first 35 keystream bits. This reduces the search space from 2^{256} to 2^{182}, giving a dimension of the pre-computation table. The main contribution of [3] is to provide state update tables which lead to the computation of the 35 output bits as part of this *conditional sampling* formulation. The paper outlines an algorithm to mount a TMDTO attack on Espresso and presents estimates of TMDTO parameters.
A recurring theme in the algebraic cryptanalysis of Espresso is *equivalence-preserving* transformations of NLFSRs in the Galois configuration to the Fibonacci configuration, or another Galois configuration with fewer feedback functions. The importance of such analysis stems from the fact that the security claims made in the proposal paper [1] against standard cryptanalytic techniques were based on a so-called 'equivalent' NLFSR which had only two non-linear feedback functions compared to the 14 feedback functions of the Espresso NLFSR. In [5], the main contribution is the formulation and analysis of a transformation which is claimed to reduce the Espresso NLFSR to an equivalent LFSR. The authors further contend that the output generated by the transformed cipher output function matches the output of the original Espresso cipher. Based on the above framework, estimates of applying standard algebraic attacks, as formulated in [6] and [7], on the equivalent LFSR-plus-non-linear output function scheme are provided.
In this article, we present an improved implementation of a *Differential Fault Injection Attack* on the Espresso stream cipher by Bathe *et al.*, reported in INDOCRYPT 2021 [2]. The concept of differential fault injection attack (cf. [8], [9]) involves the creation of additional sets of equations in the form of *difference equations* obtained from the different stages of a shift register state. These difference equations capture the cascading effect of the injection of a random fault in one of the stages. The complete set of these difference equations, generated over several clock intervals, is merged with the set of update equations for the cipher without any fault injection and the union of all these sets of equations is fed to a SAT solver. If the random fault locations have been correctly estimated using the pre-computed data, the SAT solver would return the internal state of the cipher with high probability. Differential fault attacks on Trivium, Grain, ACORN and Lizard have been reported in [10], [11] and [12]. In [2], it was reported that the injection of four random faults was enough for a successful run of this attack on the Espresso stream cipher, and the time taken for the online phase was around $400s$.

Our Contributions: We report improvements on Bathe *et al.*'s algorithm [2] in the implementation of both the pre-computation stage and the state recovery phase of the attack as follows.

1. In [2], it was suggested that the pre-computation table be constructed using 2^{20} random trials per component for all the 256 signatures. We have successfully generated the pre-computation data with a reduced number of 2^{15} **random trials per component**.

2. We have introduced an *abort-and-retry* functionality in the online phase of the state recovery attack.
3. We have succeeded in completing the state recovery attack injecting **three random faults** in comparable time with that reported in [2] using four random faults.

Organization of the paper: We begin with a brief description of the Espresso cipher and outline the concept of a differential fault attack (DFA) on a stream cipher. We then discuss the formulation and key steps of Bathe *et al.'s* differential fault attack on Espresso. Next we present the details of the improved implementation algorithm, describing first our modifications in the pre-computation phase and then in the online state recovery phase. Representative data are then tabulated for both the 4-fault injection and 3-fault injection experiments. We conclude with some observations based on our data and outline the scope of future work.

2 Differential Fault Attack on Espresso

In this section, we first briefly describe the stream cipher Espresso following [1]. Next we outline the concept of fault attack on stream ciphers from [2], and describe the different stages of such an attack, with their objectives.

2.1 The Espresso NLFSR and Output Function

We use the same notation as [1] to denote the designed NLFSR as G; the feedback function of the i-th stage of G is denoted g_i. In terms of the functions g_i, the NLFSR G is defined as follows.

$$
\begin{aligned}
g_{255}(x) &= x_0 \oplus x_{41}x_{70} \\
g_{251}(x) &= x_{252} \oplus x_{42}x_{83} \oplus x_8 \\
g_{247}(x) &= x_{248} \oplus x_{44}x_{102} \oplus x_{40} \\
g_{243}(x) &= x_{244} \oplus x_{43}x_{118} \oplus x_{103} \\
g_{239}(x) &= x_{240} \oplus x_{46}x_{141} \oplus x_{117} \\
g_{235}(x) &= x_{236} \oplus x_{67}x_{90}x_{110}x_{137} \\
g_{231}(x) &= x_{232} \oplus x_{50}x_{159} \oplus x_{189} \\
g_{217}(x) &= x_{218} \oplus x_3x_{32} \\
g_{213}(x) &= x_{214} \oplus x_4x_{45} \\
g_{209}(x) &= x_{210} \oplus x_6x_{64} \\
g_{205}(x) &= x_{206} \oplus x_5x_{80} \\
g_{201}(x) &= x_{202} \oplus x_8x_{103} \\
g_{197}(x) &= x_{198} \oplus x_{29}x_{52}x_{72}x_{99} \\
g_{193}(x) &= x_{194} \oplus x_{12}x_{121}
\end{aligned}
$$

All the remaining feedback functions are defined as: $g_i(x) = x_{i+1}$, i.e. the only input to the stage is from the preceding stage, updated with each clock. The output function is defined as:

$$\begin{aligned} z(x) =& x_{80} \oplus x_{99} \oplus x_{137} \oplus x_{187} \oplus x_{222} \oplus x_{227} \oplus x_{243}x_{217} \oplus x_{247}x_{231} \oplus x_{213}x_{235} \\ & \oplus x_{255}x_{251} \oplus x_{181}x_{239} \oplus x_{174}x_{44} \oplus x_{164}x_{29} \oplus x_{255}x_{247}x_{243}x_{213}x_{181}x_{174} \end{aligned} \tag{1}$$

Key and IV Initialization of Espresso:
Denoting the key bits as k_i, $0 \leq i \leq 127$ and IV_i, $0 \leq i \leq 95$, to be the bits of the initialization value, the loading scheme of the key and the IV bits in the shift register may be described as follows:

$$\begin{aligned} x_i &= k_i, \quad 0 \leq i \leq 127 \\ x_i &= IV_{i-128}, \quad 128 \leq i \leq 223 \\ x_i &= 1, \quad 224 \leq i \leq 254 \\ x_{255} &= 0 \end{aligned}$$

In the initialization phase, the cipher is clocked 256 times and each time the produced output bit is XOR-ed with the stages x_{255} and x_{217}. Hence, in this phase, these two feedback functions of the NLFSR are given by:

$$\begin{aligned} g_{217}^{\text{init}}(x) &= x_{218} \oplus x_3x_{32} \oplus z(x) \\ g_{255}^{\text{init}}(x) &= x_0 \oplus x_{41}x_{70} \oplus z(x) \end{aligned}$$

2.2 Bathe *et al.*'s Differential Fault Attack Strategy

A simple strategy for mounting (differential) fault attacks on stream ciphers is outlined as follows, which summarizes the technique of Bathe *et al.* in [2].

1. With a valid key and IV fed to the cipher, record a sufficiently long chunk of the keystream after initialization.
2. Then 'reset' the cipher, randomly flip one bit of the cipher internal state, and record a similar sized chunk of keystream with the same key and IV as before.
3. The two strings of keystream bits - corresponding to the *keystream without fault* and the *fault-injected keystream*, respectively - are compared. With the aid of pre-computed data and some statistical analysis of the observed keystreams, the fault location is estimated.
4. For each such bit-flip fault location estimate, a new (set of) difference equation(s) is (are) generated. Then the entire system of equations, including the update equations for the cipher without any fault injection, is fed to a SAT solver to solve for the unperturbed internal state of the cipher.

DFA on a stream cipher typically involves two phases - the *offline phase* and the *online phase*. The offline phase generates computational data, based on sufficiently large number of trials, which contain statistical information specific to the keystream generated by a fault injected in each possible location. For a stream cipher employing an n-stage feedback shift register, there are n possible fault locations. If λ keystream bits are processed for R random trials to generate the data identifying each fault, the offline computational data will be stored in a 3-dimensional array of size

$$n \times \lambda \times R$$

This offline data will be used in the first step of online calculation to extract the random fault location from an observed keystream string of length λ, employing statistical correlation tools.
Once multiple fault locations have been extracted, using multiple reset-inject-extract sequences, the attack moves into the phase where systems of equations are generated as follows.

1. The first system of equations is generated using the cipher update equations along with the cipher output equation, corresponding to the case when no fault is injected.
2. Further systems of equations are generated in the form of *difference equations* for the output and the cipher update equations, corresponding to the extracted fault locations.

The union of all the sets of equations generated as above are then fed to a SAT solver, which will return the correct internal state of the stream cipher provided that a sufficient number of correct fault locations has been used to generate the equation sets.

The Strategy of Fault Location using 'Signatures' In this subsection, following Section 3 of [2] (cf. [11] also), we present some details on the offline computation steps leading to the extraction of random faults in the initial stages of the attack.
Consider that a fault has been injected into an unknown location f of an n-stage cipher state, and we have access to $\lambda \sim n$ keystream bits. The offline phase consists of the computation of a vector of length λ, termed the *signature*, for each possible fault location (say, from $f = 0$ to $f = n - 1$) defined as follows.

$$\mathcal{S}^{(f)} = \{s_0^{(f)}, s_0^{(f)}, \ldots, s_{\lambda-1}^{(f)}\} \tag{2}$$

where $s_i^{(f)}$, corresponding to the i-th keystream bit associated with fault f is given by:

$$s_i^{(f)} := \frac{1}{2} - \Pr[z_i \neq z_i^{(f)}] \tag{3}$$

In the above equation, z_i denotes the keystream bit obtained without fault injection, and $z_i^{(f)}$ denotes the keystream bit obtained after injecting a fault in

the location f. For each possible location f of the state of the cipher, the vector $\mathcal{S}^{(f)}$, with λ components, is computed over a large number of random trials and stored in a table. The fault signature is said to be strong if the following value is close to 1.

$$\sigma\left(\mathcal{S}^{(f)}\right) = 2\sum_{i=0}^{\lambda-1} \frac{s_i^{(f)}}{\lambda} \tag{4}$$

Extraction of Random Fault Location in the Online Phase:
The online phase commences with the adversary performing ω trials, injecting a fault in the cipher in each trial, where the parameter ω is large enough to solve for the state of the cipher. The fault location for each such trial is obtained in the following steps.

1. Record the original keystream $z_0, z_1, \ldots, z_{\lambda-1}$, and then reset the cipher to the original state, re-keying the cipher with the same key and IV.
2. Inject a fault in an unknown location γ, and record corresponding keystream $z_0^{(\gamma)}, z_1^{(\gamma)}, \ldots, z_{\lambda-1}^{(\gamma)}$.
3. For each unknown fault γ, a *trail vector* is computed for $i = 0, 1, \ldots, \lambda - 1$ as:
$$\tau^{(\gamma)}[i] = \begin{cases} -\frac{1}{2}, & \text{if } z_i = z_i^{(\gamma)} \\ \frac{1}{2}, & \text{if } z_i \neq z_i^{(\gamma)} \end{cases} \tag{5}$$
4. Ideally, the fault location f for which the signature $\mathcal{S}^{(f)}$ has the closest match with the trail $\tau^{(\gamma)}$, would be the correct fault location.

In the final step, given an unknown fault index γ, one computes the sample Pearson correlation coefficient between the corresponding trail vector $\tau^{(\gamma)}$ (Equation 5) and the signature vectors corresponding to each fault location (cf. Equation 2). A list of indices, denoted $\mathcal{L}_\gamma$, is created and sorted in the decreasing order of signature-vs.-trail correlation coefficient, which eventually leads to an estimate of the fault location.

State Recovery Using the Fault-Injected Keystream Equations:

The above process is repeated ω times, for ω fault locations, hence the attack processes $\omega + 1$ keystream sequences. Denoting the fault indices as $\gamma_0, \gamma_1, \ldots, \gamma_{\omega-1}$, we have ω lists denoted by: $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_{\omega-1}$. Next choosing a set of ω fault locations, $p_0, p_1, \ldots, p_{\omega-1}$, where each $p_i \in \mathcal{L}_i$, a system of equations is solved to recover the state. The solution is used to generate a keystream of length λ, which is compared with the available keystream. If it does not match, the solution process is repeated using the next best set of locations $p'_0, p'_1, \ldots, p'_{\omega-1}$, chosen with increasing rank in the respective lists.

2.3 Setting up the Espresso Equations

Consider first the equations for the keystream bits without any fault injection. As noted in [2], the Espresso non-linear feedback shift register (NLFSR) has 14

non-linear feedback bits, for which new variables are introduced in each clocking. These are fed back into the respective positions of the state bits, and form another set of equations involving these new variables. This process results in generating equations of shorter length, which are easier to solve, at the cost of increasing the number of variables by a count of 14 every clock. This technique is described as Algorithm 3 in [2], and we present the key steps as follows.

- The output function of Espresso is defined as *eqo* and it is matched with the known keystream bits from $i = 0$ to 255.
- Then the 14 non-linear feedback functions at different intermediate stages are defined.
- Next the forward shift from the predecessor x_{i+1} to successor x_i in the shift register stages for every clocking is defined.
- *The Crucial Step:* The non-linear feedback equations are implemented by introducing new variables in every iteration precisely for the 14 stages having non-linear feedback. For instance at the i-th iteration, the existing variable x_{217} is updated with a new variable $v_{217,i}$.
- The updated variables are matched with the non-linear function values of the previous stage. This gives rise to a new set of 14 equations, labelled $E_{2,i}$ to $E_{15,i}$, in every iteration.
- Finally the set of equations Eq is updated by considering the union with the set $Eq_{1,i}$, for the output function matched with the i-th keystream bit, and the 14 equations described earlier.

Generating Bit Difference Equations for Fault-injected Keystream If a fault is injected at f in the state of the cipher, the change in the values of the state bits is:

$$\Delta x_i = \begin{cases} 0, & \forall i \in \{0, 1, \ldots, f-1, f+1, \ldots, 255\} \\ 1, & i = f \end{cases}$$

To obtain new sets of equations, difference equations are computed using the output equations and the non-linear update equations. For instance, the difference expression corresponding to the feedback function at stage 217, given by $f_{217} = x_{218} \oplus x_3.x_{32}$, is shown below.

$$\Delta f_{217} = \Delta x_{218} \oplus \Delta x_3.x_{32} \oplus x_3.\Delta x_{32} \oplus \Delta x_3.\Delta x_{32}$$

where Δx stands for the difference between the initial and final values of some Boolean variable x.
A brief description of the key steps of the algorithm for generating the difference equations for the fault-affected keystream bits from Section 4 of [2] follows.

1. Given the fault location f, the difference is initialized as $\Delta x_f \leftarrow 1$.
2. Then the difference in the output at the i-th keystream bit, Δz_i, is computed.

3. Next, the difference equations corresponding to the 14 non-linear update functions are computed.
4. The following additional equation is appended to the update equation set:

$$I = \Delta z_i \oplus z_i \oplus z_i^f \tag{6}$$

   This matches the computed output difference Δz_i with that between the i-th keystream bit and the fault-affected i-th keystream bit.
   The set of equations is updated as $Eq = Eq \cup I$.
5. Then the *differences* at the various stages are updated taking into account the feedback difference equations computed for the i-th iteration.
6. The final steps use the algorithm for bits without fault injection to update the state bits for the i-th iteration.

Hence, given an extracted fault location $f \in \{0, 1, \ldots, 255\}$, the difference equation for the i-th keystream bit captures the effect of the fault in the f-th stage of the state. The propagation of the fault introduced in the state f will be captured by the updates of the difference equations for all the non-linear states in the subsequent steps.
The sole equation generated for the solver pertaining to a fault injected at f is given by Equation 6, computed before the update steps, for each keystream bit observed. Obviously, for correct fault extraction, the *computed difference* Δz_i will be equal to the difference $z_i \oplus z_i^f$ of the observed bits for the fault-free and fault-injected strings. Therefore, if the experiment processes λ generated bits for m injected faults, the solver will be fed an additional $m\lambda$ equations, for a total of $(15 + m)\lambda$ equations including the fault-free equation set.

3 An Improved Differential Fault Attack on Espresso

In what follows, we discuss the different stages of our version of the differential attack on Espresso. As stated before we closely follow the steps described in [2], but with some key modifications introduced by us in our implementation.

3.1 Offline Phase: Experimental Observations

Recall, from the discussion in the previous subsection, that the offline phase involves the computation of the signature vectors of length λ, λ being the number of observed keystream bits, for every possible fault location. Our modification of the implementation of the offline phase was based on the following observations.

1. While [2] suggests $\lambda \sim n = 256$, the paper actually presents pre-computation results for $\lambda = 100$, and claims to use 2^{20} trials per component for all the 256 signatures. Our efforts, using a system with better specifications than those described in [2], to simulate for $\lambda = 256$ and 2^{20} trials did not produce any output over a duration of several days.

2. Reducing the number of trials per data point to 2^{10} yielded quick results, but the signature estimates did not exhibit satisfactory statistical characteristics.
3. Increasing the number of trials to 2^{12}, while reducing λ to $100 < n/2$, yielded better estimates.
4. The best estimates, with 2^{15} random trials per component, which turned out to be adequate for successful attacks, were obtained using a *parallel processing approach* detailed next.

3.2 Using Multiple Cores to Speed Up Random Trials

We implemented the offline computation using SageMath 9.5 on HP `Z820` desktop Workstation (Intel Xeon(R) CPU `E5-2630` 0 @ 2.30GHz × 24, 96 GiB RAM, Ubuntu-22.04.4).
To compute a signature vector for a specific fault location, we executed the procedure (cf. Section 2.2) with 2^{15} random trials for each component, each time with a randomly generated Key (128-bit) and a randomly generated IV (96-bit). We parallelized the computation with the following steps:

1. Obtain information about the number of cores available in the system.
2. Divide the task equally among all possible cores (i.e. 256 divided by the number of cores). For example, if there are 16 available cores, core 1 executes the signature vector for fault locations 0-15, core 2 executes the signature vector for fault locations 16-31, and so on.
3. Collect the outputs executed by the individual cores and combine them to obtain the desired result.

The above procedure was implemented with a shell script and has yielded signature vectors leading to successful instantiations of the fault attack on Espresso with high probability.

Choice of Functions in Signature Computation and Random Fault Extraction We briefly outline some of the *optimized choices* in the selection of statistical functions among the several available packages.

- *Choice of the Correlation Function:*
  We chose to implement our correlation analysis with the function `pearsonr` from the **scipy.stats** package instead of the function `pearson_correlation` from the **sage.stats.correlation** package. The reasons for the above choice, based on experimental evidence, are as follows:
  1. The `pearson_correlation` was shown to consume significantly more time than the function `pearsonr`.
  2. The output of the `pearson_correlation` function required further processing to be fed as input in the subsequent stages of the algorithm.

– *Choice of the random number generator:*
We have further optimized our implementation with a judicious choice of the random number generator, using `randrange` and `randbits`, depending on the requirement of the algorithm.
Specifically, to generate the key and IV of the cipher we used the function `randbits` from the **secrets** package, to specify the exact number of bits in the representation of the generated random number.
Unspecified random numbers were generated with `randrange` from the **random** package, which returns a random integer in the range specified.

3.3 Online State Recovery: Abort-and-Retry Implementation

As discussed earlier, we have implemented the attack in three stages as follows:

1. Generation of offline data and extraction of random faults using the offline data.
2. Generation of the sets of equations for the fault-free keystream and the difference equations for the fault-affected keystreams.
3. Internal state recovery of Espresso stream cipher using the SageMath SAT solver.

Our modification, pertaining to this phase of the implementation, stemmed from the following observation. In the online phase, ascertaining whether the faults have been correctly extracted can only be done depending on the SAT solver output. Two situations may arise where the attack is deemed unsuccessful. In the first, the SAT solver fails to produce a solution within a predetermined *timeout period*, which is based on the proof-of-concept estimates of the run-time of successful attacks. Or, the SAT solver produces a solution before time-out, but the output keystream based on the recovered state does not match with the observed initial output.
In either case, we revert to the fault extraction stage by choosing a new set of fault position candidates, based on their Pearson correlation scores, according to the algorithm. Our implementation of the online phase features this **abort-and-retry** functionality, which stops the SAT solver and goes back to retrieve a new set of fault locations to resume the attack. This has been achieved by implementing a **multiprocessing routine** in carrying out the complete attack as described below.

Multiprocessing Implementation of Abort-and-Retry If the extracted fault locations do not match the actual random ones, the SAT solver output may be delayed indefinitely or be erroneous. To address this issue, we have implemented a sequence where the SAT solver is terminated by a *parent process* after the solver has run for a *timeout period*. The timeout period is based on the available run-time estimates of successful attacks. Then a new set of fault locations is chosen, having the next highest correlation score, to prepare the equations anew to be fed to the SAT solver.

The Python *multiprocessing package* has been used to call the SAT solver module as a *child* of the parent process. However, to ensure that the parent process can access a successful output of the SAT solver process, we have implemented an *interprocess communication (IPC)* protocol.
Two standard techniques exist in Python to implement IPC, namely, `queue` and `pipe`; we have implemented IPC for the DFA attack using `queue`. In the function definition, we have appended an extra variable `result_queue` where the solver process writes the obtained result for a successful run, which is then shared with the parent process.
The entire procedure is outlined in the flow diagram of Figure 1.

Validation of Fault Location Update Strategy In implementing the abort-and-retry via multiprocessing when the SAT solver failed to return a valid output within the timeout period, we adopted the following strategy for updating the fault location candidates before the solver re-run. We only swapped a single fault location index with the index having the highest Pearson correlation score among those remaining. Recall that this score was obtained using the pre-computed signature data and the trail data of the fault locations computed online. If that did not produce a desired output, our implementation code declared that the attack had failed.
We provide the following experimental validation of the above strategy by computing the frequencies of failure in extracting the correct fault location for up to a single swap of indices based on the correlation score. Over 32768 trials, the initial choice identified the correct fault location with probability 0.96; allowing a single swap increased the probability of successful identification to 0.99.
This also provides an indirect verification of our pre-computed signature data which was obtained using a different approach, requiring significantly less computation, than that of the original paper.
The Pearson correlation scores of the indices were stored as a sorted list in our implementation, so that the first index corresponds to the highest score, the second index corresponds to a score less than or equal to the first and so on, in a monotonically decreasing fashion.

Experimental Validation of Fault Update Strategy

- Number of Trials: 32768; Execution Time: $\sim$ 2471.91 Seconds.
- Number of correctly identified fault locations at the first index of the sorted index list: 31409.
- Number of correctly identified fault locations at the second index of the sorted index list: 949.
- Probability of SUCCESS in identifying fault location at the first index of the sorted index list: 0.96.
- Probability of SUCCESS in identifying fault location within the first two indices of the sorted index list: 0.99.

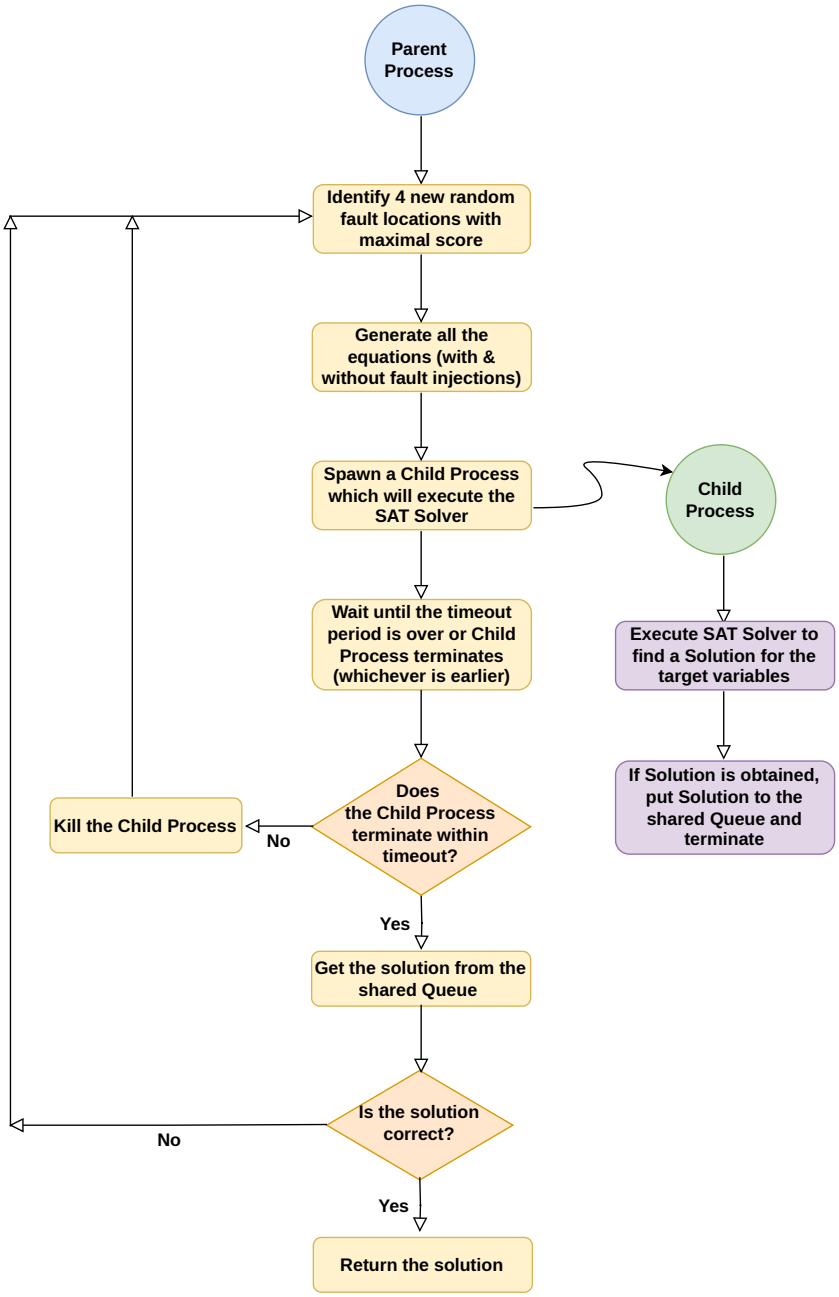

Fig. 1. Flowchart of Multiprocessing in DFA Attack

4 Experimental Results

The algorithms described in [2] were implemented by Bathe *et al.* using SageMath [13] and the Sage SAT solver on a system with a 2208MHz processor, 8 GB RAM running on Ubuntu 20.04. It was claimed that *injecting four random faults is enough for a SAT solver to correctly extract the values of all the state bits.* They reported that injecting faults at four random locations $\{212, 33, 155, 3\}$ and using 200 keystream bits to generate a system of equations for the SageMath solver returned the correct state bits in approximately 400 s.
We first append a sample of our experimental results tabulating 4-tuples of random faults injected and the time taken for the algorithm to recover the internal state. Next we present the same data for successful state recovery attack using **three randomly injected faults**. Our experiments were performed on a system with 16 GB RAM and a 12th Gen i7-1255U processor with 12 cores, running on Ubuntu 22.04.

4.1 Experimental Results for 4 Fault Injection DFA

In the following Table 1, we present representative data of successful state recovery using four randomly injected faults.

Table 1. 4-Fault Injection DFA

Random Fault Locations	**Solution Time (seconds)**
[209, 16, 128, 87]	409.87
[134, 133, 163, 217]	312.94
[171, 103, 209, 115]	517.29
[220, 199, 17, 24]	441.70
[111, 210, 144, 214]	295.22
[209, 177, 199, 84]	302.48
[177, 169, 179, 186]	274.77
[214, 95, 212, 107]	297.03
[60, 163, 218, 29]	452.12
[179, 79, 68, 175]	349.60

We note that while the original paper reported a state recovery time of $\sim 400s.$, our sample experiments tabulated above succeeded with an average time of **365.30**s., with a fastest run of **274.77**s.

4.2 Experimental Results for 3 Fault Injection DFA

We have demonstrated internal state recovery of the Espresso stream cipher by differential fault injection attack **using three faults instead of four**, as pro-

posed in [2]. Displayed below is the experimental data corresponding to a subset of the numerous successful instances of the aforesaid attack, which demonstrate attack timings comparable to the four-fault injection experiments.

Experimental Data: Contiguous Three-Fault Injection DFA In the following Table 2 we display representative data for differential fault injection attacks, using only three fault locations, which appear to be closely spaced.
It is evident that all the instances share the first two indices given by $171, 187$*, and small changes in the third index continue to provide optimum fault locations for the attack.*
Implementation Setup: The codes were implemented on a computer system equipped with a 12th Gen i7-1255U processor having 12 cores, along with 16 GB of RAM, operating on Ubuntu 22.04, using SageMath version 9.5.

Key : 0X9510592363D65E8EB89AECF55771ECB4
IV: 0XADAF4727BE237632424F3B2F

Table 2. Contiguous 3-Fault Injection DFA

Random Fault Locations	**Solution Time (seconds)**
$[171, 187, 207]$	393.62
$[171, 187, 208]$	835.47
$[171, 187, 211]$	894.56
$[171, 187, 214]$	209.19
$[171, 187, 215]$	154.00
$[171, 187, 216]$	512.00
$[171, 187, 218]$	631.16
$[171, 187, 225]$	576.49
$[171, 187, 229]$	156.69
$[171, 187, 231]$	316.15
$[171, 187, 236]$	147.16
$[171, 187, 249]$	124.48

Experimental Data: "Random" Three-Fault Injection DFA For the sake of completeness, we also append representative data of successful DFA attack using three random fault locations, which are farther apart and do not exhibit any noticeable pattern in Table 3.

Key : 0X9510592363D65E8EB89AECF55771ECB4
IV: 0XADAF4727BE237632424F3B2F

Table 3. Non-Contiguous 3-Fault Injection DFA

Random Fault Locations	**Solution Time (seconds)**
[165, 203, 189]	159.22
[98, 170, 111]	327.86
[176, 130, 178]	343.28
[157, 93, 212]	370.66
[172, 168, 165]	410.05
[183, 79, 202]	426.10
[246, 230, 148]	474.19
[4, 236, 185]	527.33
[103, 143, 174]	531.71
[237, 82, 160]	546.68
[131, 121, 219]	767.96
[125, 194, 157]	833.88

Table 4 provides statistical data pertaining to the 3-fault injection experiments. It summarizes key metrics associated with the time taken by the SAT Solver to provide a correct solution, such as Average, Median, Standard Deviation, Maximum and Minimum, derived from 153 data points. Each data point corresponds to a different 3-fault location leading to a successful attack. It is noted that the majority of the 3-fault attacks succeeded in times comparable to 4-fault attacks, but the average time obtained from our data is skewed by the presence of a few extremely lengthy experiments (maximum time exceeding 9.5 hours). In fact the quickest successful 3-fault attack took only 124.48 seconds, which was better than what we have recorded for our 4-fault experiments.

Table 4. 3-Fault Injection DFA Statistics

Number of Data Points	153
Average	1597.55 Seconds
Median	669.80 Seconds
Standard Deviation	3246.35 Seconds
Max	34458.49 Seconds
Min	124.48 Seconds

5 Conclusion and Future Work

In this article we have presented an improved implementation of a differential fault attack on Espresso, based on the work of Bathe *et al.* in [2]. We have speeded up the offline computation of the fault signatures, by **conducting 2^{15} random trials per element instead of 2^{20} random trials** prescribed by the previous work. Further, we have implemented an original **abort-and-retry functionality** with a multiprocessing routine, in the online phase of the attack. These tweaks, coupled with a judicious choice of parameters and functions have resulted in the following significant development:
Successful state recovery attacks were mounted in comparable time using only three fault injections instead of four as reported in the existing work.
An inspection of our experimental data, as presented in Section 4, leads to the question whether it is possible to further reduce the number of random fault injections and still mount a successful state recovery attack in reasonable time. Also, further analysis of our data on the *contiguous* 3-fault locations may possibly reveal some dependencies among the stages involved, which can be exploited to improve algebraic attacks on Espresso.

Acknowledgement The authors would like to thank the anonymous reviewers for their detailed comments that improved the technical content as well as the presentation of the paper. The inputs from Ravi Anand, Mridul Nandi and Subhamoy Maitra are also gratefully acknowledged. Sabyasachi Karati acknowledges the support of SAG, DRDO, related to the project “Analysis of Cryptographic Algorithms and Protocols used in Wireless Mobile Communications" as well as the support of MeitY, Government of India, related to the initiative “Cluster - Cryptography, Information Security Education and Awareness (ISEA) Project Phase - III".

References

1. Dubrova, E., and Hell, M.: Espresso: A stream cipher for 5G wireless communication systems. Cryptography and Communications.**9**, 273–289. Springer(2017). https://doi.org/10.1007/s12095-015-0173-2
2. Bathe, B., Tiwari, S., Anand, R., Roy, D. and Maitra, S.: Differential fault attack on Espresso. INDOCRYPT 2021. 271–286. Springer (2021). https://doi.org/10.1007/978-3-030-92518-5_13
3. Sinha, N.: Internal state recovery of espresso stream cipher using conditional sampling resistance and TMDTO attack. Advances in Mathematics of Communications. **15**, 539–556. AMC(2021). https://doi.org/10.3934/amc.2020081
4. Maitra, S. and Sinha, N. and Siddhanti, A. and Anand, R. and Gangopadhyay, S.: A TMDTO Attack Against Lizard. IEEE Transactions on Computers. **67**, 733-739. AMC(2018) https://doi.org/10.1109/TC.2017.2773062
5. Yao, G. and Parampalli, U.: Generalized NLFSR transformation algorithms and cryptanalysis of the class of espresso-like stream ciphers. arXiv preprint arXiv:1911.01002 (2019). https://doi.org/10.48550/arXiv.1911.01002

6. Courtois, N. and Meier, W.: Algebraic Attacks on Stream Ciphers with Linear Feedback. In: Biham, E. (ed.) EUROCRYPT 2003. Advances in Cryptology, LNCS, vol. 2656, pp. 513-525. Springer, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_21
7. Rønjom, S. and Helleseth, T.: A New Attack on the Filter Generator. IEEE Transactions on Information Theory. **53**, 1752-1758. (2007). https://doi.org/10.1109/TIT.2007.8946
8. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski, B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513-525. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0052259
9. Biham, E., Dunkelman, O.: Differential cryptanalysis in stream ciphers. https://eprint.iacr.org/2007/218.pdf
10. Hojsík, M., Rudolf, B.: Differential fault analysis of trivium. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 158-172. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71039-410
11. Sarkar, S., Dey, P., Adhikari, A., Maitra, S.: Probabilistic signature based generalized framework for differential fault analysis of stream ciphers. Cryptogr. Commun. 9(4), 523–543 (2017). https://doi.org/10.1007/s12095-016-0197-2
12. Siddhanti, A., Sarkar, S., Maitra, S., Chattopadhyay, A.: Differential fault attack on grain v1, ACORN v3 and lizard. In: Ali, S.S., Danger, J.-L., Eisenbarth, T. (eds.) SPACE 2017. LNCS, vol. 10662, pp. 247–263. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71501-814
13. The Sage Developers. SageMath, the Sage Mathematics Software System (Version 9.0+), (2020). https://www.sagemath.org