ToFA: Towards Fault Analysis of GIFT and GIFT-like Ciphers Leveraging Truncated Impossible Differentials

Anup Kumar Kundu[1], Shibam Ghosh[2,3], Aikata Aikata[4] and Dhiman Saha[5]

[1] Indian Statistical Institute, Kolkata 700108, India
anupkundumath@gmail.com
[2]Department of Computer Science, University Of Haifa, Haifa, Israel
[3]Inria, Paris, France
shibam.ghosh@inria.fr
[4]Institute of Information Security,
Graz University of Technology, Austria
aikata@tugraz.at
[5]de.ci.phe.red Lab, Department of Computer Science & Engineering,
Indian Institute of Technology Bhilai, Chhattisgarh - 491002, India
dhiman@iitbhilai.ac.in

Abstract. In this work, we introduce ToFA, the first fault attack (FA) strategy that attempts to leverage the classically well-known idea of impossible differential cryptanalysis to mount practically verifiable attacks on bit-oriented ciphers like GIFT and BAKSHEESH. The idea stems from the fact that *truncated* differential paths induced due to fault injection in certain intermediate rounds of the ciphers lead to active SBox-es in subsequent rounds whose inputs admit specific truncated differences. This leads to a (multi-round) impossible differential distinguisher, which can be *incrementally* leveraged for key-guess elimination via partial decryption. The key-space reduction further exploits the multi-round impossibility, capitalizing on the relations due to the quotient-remainder ($\mathcal{QR}$) groups of the GIFT and BAKSHEESH linear layer, which increases the filtering capability of the distinguisher. Moreover, the primary observations made in this work are independent of the actual SBox. Clock glitch based fault attacks were mounted on 8-bit implementations of GIFT-64/GIFT-128 using a ChipWhisperer Lite board on an 8-bit ATXmega128D4-AU micro-controller. Unique key recovery was achieved for GIFT-128 with 3 random byte faults, while for GIFT-64, key space was reduced to 2^{32}, the highest achievable for GIFT-64, with a single level fault due to its key-schedule. To the best of our knowledge, this work also reports the highest fault injection penetration for any variant of GIFT and BAKSHEESH. Finally, this work reiterates the role of classical cryptanalysis strategies in fault vulnerability assessment by showcasing the most efficient fault attacks on GIFT.

Keywords: Fault Analysis, Impossible Differential, GIFT, BAKSHEESH

1 Introduction

Fault analysis has attained a top spot in the list of physical attacks over time with the Differential Fault Analysis (DFA) being the most widely used variants. Beyond the basic power of fault injection, DFA generally benefits from the classical differential cryptanalysis

(DC) paradigm. DFA can essentially be visualized as the round-reduced DC of a cipher being applied to its full-round version by using the fault as the source of inducing the input difference to the differential trail. The basic difference is that while classical DC is a Chosen Plaintext Attack (CPA), DFA actually relies on the fault model to determine what kind of intermediate differences can be induced and can be referred to as a Restricted Intermediate Difference Attack (RIDA). As the nature of the intermediate difference is dictated by the fault model, it plays a very important role in making physical attacks like fault attacks realistic.

Since the introduction of fault analysis in public key cryptography (RSA-CRT implementations by Boneh et al. [BDL97] [BDL01]) and subsequently in symmetric cryptography (DFA on DES by Biham and Shamir [BS97]), the idea has been extended along various verticals. One vertical constitutes the nature of the faults and how it is interpreted, viz., Statistical Fault Analysis (SFA) [FJLT13], Collision Fault Analysis (CFA) [BK06], Persistent Fault Analysis (PFA) [ZLZ+18], Statistically Ineffective Fault Analysis (SIFA) [DEG+18], and Statistically Effective Fault Analysis (SEFA) [VZB+22]. On the other vertical we can find classical cryptanalysis being used for developing new types of fault attacks, viz., Integral Fault Analysis (IFA) [SSL15], Slow Diffusion Fault Analysis (SDFA) [DP20, AKS20, KAKS22], Internal Differential Fault Analysis (InDFA) [SC16], Meet-in-the-Middle and Impossible Differential Fault Analysis (ImpDFA) [DFL11]. The current work is an attempt to add to the state-of-the-art in the latter vertical while reporting the most efficient fault attacks on GIFT exploiting impossible differential trails. It is worth mentioning that though the current work constitutes an application of impossible differentials in FA, it not the first such attempt. The first instance of this is attributed to Derbez *et al.* for their fault analysis on AES [DFL11].

The block cipher GIFT [BPP+17] proposed by Banik *et al.* in CHES 2017, is an excellent attempt at achieving lightweightness where the design exploits the interaction of both the SBox and linear layers to beat its predecessor PRESENT [BKL+07] in terms of ASIC chip-area as well as latency. The interest in GIFT has been recently renewed after the lightweight authenticated cipher GIFT-COFB [BCI+20] moved to the final round of the NIST Lightweight Cryptography Competition [Tec17]. The authors of GIFT introduce BOGI (Bad Output must go to Good Input) permutation which can be interpreted as the concatenation of 16-bit sub-permutations using what is referred to as quotient-remainder ($\mathcal{QR}$) groups. GIFT has two variants namely, GIFT-64 and GIFT-128 which support state-sizes of 64 and 128 bits with 28 and 40 rounds respectively, however, with the same key-size of 128-bits. It is interesting to see that despite being outdated, there seems to be only a *few* efforts in the FA of GIFT. This constitutes one of the primary motivations of the current work. A multi-round fault based DFA was reported by Luo *et al.* on GIFT-64 by injecting random nibble faults in round 27, 26, 25, and 24 in a phased manner to recover the master key [LCMW21]. It is worth noting that this DFA **does not** exploit the permutation layer of GIFT and merely recovers round-keys incrementally injecting faults in preceding rounds while *peeling* the cipher with every recovered round-key. Min *et al.* [MFJ21] used a random nibble fault attack on GIFT to recover the master key using 31 and 32 faults on GIFT-64 and GIFT-128 respectively. Liu *et al.* [LGH22] present a fault attack on authenticated modes of GIFT. Both the works rely on peeling the cipher since faults injected at a single round are insufficient for their success. In this work, we avoid this *peeling* by leveraging the permutation layer and hence need faults to be injected in *only* one round. This also makes strategy more attacker-friendly since the fault injection set-up required is simpler and would require syncing/triggering once in the temporal domain while injecting the fault. Moreover, reliance on solely the random byte fault model makes it more attractive in the real-world realization perspective.

To emphasize on the versatility of the attack strategy devised here, we also target the very recently proposed lightweight cipher BAKSHEESH [BBC+23] by Bakshi *et al.* which

claims to be an improvement over GIFT. It has a state-size of 128-bits and borrows the BOGI permutation from GIFT, differing from GIFT in the SBox, number of rounds (35 instead of 40), key-schedule and the round-key addition which adds the round key to the entire state. An interesting aspect of the design of BAKSHEESH SBox is the presence of a *linear structure* (LS). In a recent work, Jana *et al.* [JKP24] mounted a DFA on BAKSHEESH using random *but known* bit faults in R^{32}. Based on the knowledge of location and value of the induced faults they leveraged three round deterministic trails using the LS in BAKSHEESH SBox and uniquely recovered the key space using 12 faults. The current work, as stated earlier, uses a random-byte fault model without needing the value or location of the fault while making a full key-recovery with only 4 faults. A potential target for ToFA could be DEFAULT [BBB+21] given its resemblance to GIFT albeit with a different SBox. It was claimed to be fault-resistant by design owing to the linear structures in the SBox. However, in this work we have chosen to exclude DEFAULT as a target due to reasons discussed in Section 7. We now briefly enumerate the contribution of this work.

1.1 Our Contribution

Truncated Differential Based Fault Invariants The primary contribution constitutes the identification of truncated differential trails to create fault-invariants. The main observation is that for GIFT-64, clusters of two round truncated differential trails exist that exhibit invariant properties in terms of the permissible *patterns* of the entire output truncated difference.[1] Moreover, for GIFT-128 and BAKSHEESH we observe clusters of three round truncated differentials that exhibit invariant truncated differentials. These properties are exploited as fault invariants, when random byte faults are injected in an intermediate round of the cipher. The byte-faults induce state differences that conform to the input truncated difference thereby leading to restricted patterns after two/three rounds. Any pattern that is beyond this set of permissible patterns implies an impossible differential distinguisher and can be leveraged to eliminate key candidates after partially decrypting the faulty/fault-free ciphertexts.

Multi-round Impossible Truncated Differential Filter in the Secret SBox Model The second contribution lies in turning the above distinguisher into a multi-round elimination filter which fully exploits the GIFT/ BAKSHEESH construction. The ability to invert multiple rounds with partial key guess is facilitated by the quotient-remainder ($\mathcal{QR}$) groups in the BOGI permutation of GIFT/ BAKSHEESH. It can be noted that these groups are independent and hence two-rounds can be visualized as a concatenation of four (for GIFT-64) or eight (for GIFT-128/ BAKSHEESH) $\mathcal{QR}$ groups. Guessing the partial round-key separately for each such group for Round r and Round $(r-1)$, we can verify if the intermediate state difference at the input of Round $(r-1)$ admits an *impossible truncated difference*. This filter is further augmented by combining (computing the Cartesian product) the reduced key-spaces pertaining to consecutive $\mathcal{QR}$ groups and re-verifying the corresponding truncated patterns at the Round $(r-1)$. We repeat this process while doubling the number groups used for the Cartesian product of the key-spaces until all the groups are involved at which point we verify the complete truncated pattern at Round $(r-1)$ input. This incremental procedure drastically reduces the key-space which can be propagated backwards to leverage the expected truncated pattern in Round $(r-2)$.

For 128-bit variant of the ciphers the pattern is exploitable until Round $(r-3)$. This gives us a multi-round impossible truncated differential distinguisher that fully

[1] It is worth noting that Min *et al.* [MFJ21] make a some-what similar observation of two round truncated differential but do not seem to fully exploit this and thus have to resort to *peeling* the cipher to recover master key, making their work an instance of multiple round fault attack.

exploits the round function. Combining all these we propose ToFA which is the first multi-round impossible (truncated) differential FA on GIFT and BAKSHEESH. Further, the impossibilities leveraged in this work are independent of the actual SBox and essentially exploit the bit permutation of GIFT. In case of BAKSHEESH, further reduction in key-space is done using the key-schedule. Here, we would like to emphasize that the design decision of BAKSHEESH to use add-round-keys to the entire state makes the initial key-space quadratically bigger than GIFT-128 making the attack difficult. On the other hand, the simpler key-schedule ends up making the attack easier as it works like a secondary filter to reduce the key-space. Figure 1 gives the high-level abstraction of ToFA. The first plane on the left shows the fault induced truncated difference which diffuses as a multi-round truncated differential. The cipher-text plane shows the evolution of the trail in the reverse direction with partial key-guess. The intermediate planes show multiple opportunities of key-guess eliminations.

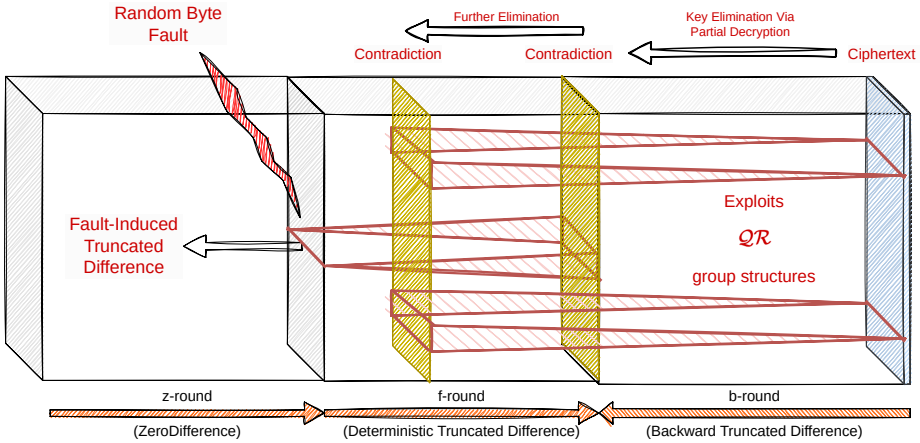


Figure 1: Overview of ToFA. The three vertical squares illustrate the structural bifurcation of the cipher rounds. The lower arrows indicate the properties that emerge before and after fault injection. The intermediate planes highlight the contradiction between forward and backward truncated differences, which we leverage to filter the round keys.

Fault Penetration The third contribution is in terms of fault penetration. Fault penetration (FP) refers to the earliest round (counting from the last) in which a fault is induced during an attack. It is well established in the literature that the higher the FP, the more challenging the attack becomes due to fault diffusion. For example, if key recovery requires injecting faults in the 8th round of a cipher (e.g., AES-128), the FP is $10-8=2$. Another attack targeting the 7th round has FP=3, making it more effective. Most existing fault attacks fail if the fault is injected earlier than the round specified by the authors. On the other hand, a higher FP implies that more rounds require protection. From a fault attack countermeasure perspective, there is an inherent trade-off between the number of protected rounds and the associated overhead. As a result, designers typically limit protection to rounds where successful state-of-the-art FA techniques have been demonstrated against the cipher.

Given these challenges, developing attacks with a higher FP is difficult and significantly impacts countermeasure strategies. This is particularly relevant for lightweight ciphers such as GIFT. Prior to our work, the highest recorded FP for a fault attack on GIFT-64/128 was 2. In our attack, for r round GIFT-64 variant, the faults can be injected as early as in round $(r-4)$, while for r round GIFT-128/BAKSHEESH, the faults can be injected as early as in round $(r-5)$. This makes the attacks reported here most efficient in

terms of fault penetrations as we achieve an FP of 4 for GIFT-64 and an FP of 5 for GIFT-128/BAKSHEESH.

Experimentally Verified Practical Key-space Reduction All claims are substantiated by real-world fault attacks on 8-bit implementations of GIFT and BAKSHEESH on the ATXmega128D4-AU micro-controller with random byte faults. The fault-injection framework constitutes clock-glitching using the ChipWhisperer Lite (CW1173) board. We also furnish a fault profiling of the ATXmega128D4-AU micro-controller by injecting faults in the last round and characterize the fault distribution by observing the ciphertext differences to establish the effectiveness of the random byte model.

Detailed experimental results give empirical evidence confirming the theoretical claims of ToFA. ToFA on GIFT-64 results in a reduced key-space of 2^{32} with 22 random byte faults. It is important to note that due to the key-schedule of GIFT-64, 2^{32} is the maximum reduction theoretically possible for the 128-bit key with random byte faults exploiting only truncated differential patterns across three rounds. ToFA on GIFT-128 recovers the unique 128-bit master key with 3 random byte faults while 2 faults give a reduction of $2^{128} \rightarrow 2^3$. For BAKSHEESH we retrieve the full key with 4 faults. To the best of our knowledge, the figures reported here are the best across literature considering that we are able to recover the master-key while targeting just a single round and hence do **not** need to *peel* the cipher while recovering individual round-keys. Table 1 summarizes the results reported in this work.

Table 1: Summary of fault attacks on GIFT and BAKSHEESH

Primitive	Fault Model	#faults	FI Point⋆	Key-Space	Ref	Remarks
GIFT-128	Random Byte	3 2	35	1 2^3	Sec. 6	Single-Round Fault
	Random Nibble	32	38	2^{17}	[MFJ21]‡	Multilevel Fault†
GIFT-64	Random Byte	22	24	2^{32}	Sec. 6	Single-Round Fault
	Random Nibble	31	26	2^{16}	[MFJ21]‡	Multilevel Fault†
		64	$27 \rightarrow 26 \rightarrow 25 \rightarrow 24$	$2^{11.91}$	[LCMW21]‡	Multilevel Fault†
BAKSHEESH	Random Byte	4	30	1	Sec. 4.2‡	Single-Round Fault
	Random but Known Bit	12	32	1	[JKP24]‡	

† Require multiple round faults due to reliance on *onion peeling* strategy for master key recovery.
⋆ Number of rounds in GIFT-64, GIFT-128, and BAKSHEESH is 28, 40, and 35 resp. ($0 \rightarrow 27/39/34$).
‡ Based on simulated FI.

Organization of the paper In Section 2, we define the notation for the paper as well as an overview of the targeted ciphers GIFT and BAKSHEESH. Then, in Section 3, we introduce ToFA as our main contribution. Section 4 represents the applicability of ToFA on the ciphers GIFT-128 and BAKSHEESH. In Section 5 we apply the same technique on GIFT-64. The experimental results along with practical verification of the attack is discussed in Section 6. Some countermeasures are also detailed. Finally, after giving a brief discussion on the cryptanalytic strategies of both the ciphers in Section 7 we conclude the paper in Section 8.

2 Preliminaries and Background

2.1 Notations

In this section we introduce the notations which we intend to use to illustrate the properties exploited to mount the fault attacks described later. We use bold lowercase letters to represent vectors in a binary field.

- For any n-bit vector $\mathbf{x} \in \mathbb{F}_2^n$, its i-th coordinate is denoted by x_i, thus we have $\mathbf{x} = (x_{n-1}, ..., x_0)$.
- $\mathbf{0}$ represents the binary vector with all elements being 0.
- The Hamming weight of $\mathbf{a} \in \mathbb{F}_2^n$ is $\mathcal{H}(a) = \sum_{i=1}^{i=n} a_i$
- For any bit-vector $\mathbf{a}$, we often call the bit positions with value 1 as the *active* bit positions.
- In this paper, R^ℓ denotes the ℓ-th round function of an r-round iterative cipher for $\ell = 0, 1, ..., r-1$ and consequently $R^{-\ell}$ denotes inverse of the ℓ-th round function.
- The input state and round key of R^ℓ are denoted by S^ℓ and K^ℓ, respectively.
- In the current work, we concentrate on GIFT-like ciphers, which have 4-bit SBox-es and hence a nibble based representation. In this regard, the i-th nibble of S^ℓ is denoted as N_i^ℓ and the j-th bit of this nibble is denoted as $N_{i,j}^\ell$.
- In terms of endianness, the right most nibble is considered as the 0-th nibble while the right most bit is considered as the 0-th bit.
- For two n-bit vectors u and v, we call u as the *successor* of v denoted as $u \succeq v$ if $u_i \geq v_i$ for all $i = 0, 1, ..., n-1$. Alternatively, v is referred to as the *predecessor* of u.
- For any n-bit vector $\mathbf{x} \in \mathbb{F}_2^n$, k-bit circular right shift and k-bit circular left shift are denoted by $x \ggg k$ and $x \lll k$, respectively. For, e.g., $[00110] \ggg 2 \rightarrow [10001]$.

2.2 GIFT [BPP+17]

The GIFT family of block ciphers is designed to prioritize hardware efficiency while maintaining a high level of security. The family includes two versions: GIFT-64 and GIFT-128. Both the versions utilize a 128-bit key, but they differ in terms of state size and the number of rounds. The GIFT-64 cipher consists of 28 rounds and has a state size of 64 bits, while GIFT-128 incorporates 40 rounds and features a larger state size of 128 bits. Each round of both ciphers comprises four operations: the SBox-Layer, Bit Permutation Layer, Addition of round keys, and Addition of Round Constants.

At the beginning of the encryption process, the cipher takes a 128-bit master key and generates round keys for each round. For GIFT-64, each round key is 32 bits, whereas for GIFT-128, the round keys are 64 bits. Within each round, the 4-bit SBox operation is applied to each nibble of the state bits. The resulting bits are then permuted through the permutation layer. Next, the round keys and round constants are XOR-ed with specific positions within the state. For GIFT-64, the round keys are XOR-ed with the 0-th and 1-st bit of each nibble, while for GIFT-128, the round keys are XOR-ed with the 1-st and 2-nd bit of each nibble. Additionally, 6-bit round constants are XOR-ed with specific positions within the state in both versions of the cipher.

In the key schedule process, depending on the version, two or four 16-bit words are extracted from the key state to generate a round key. Some key words undergo circular rotation to produce the next round keys. In round R^{i+1}, different 32 or 64-bit words are

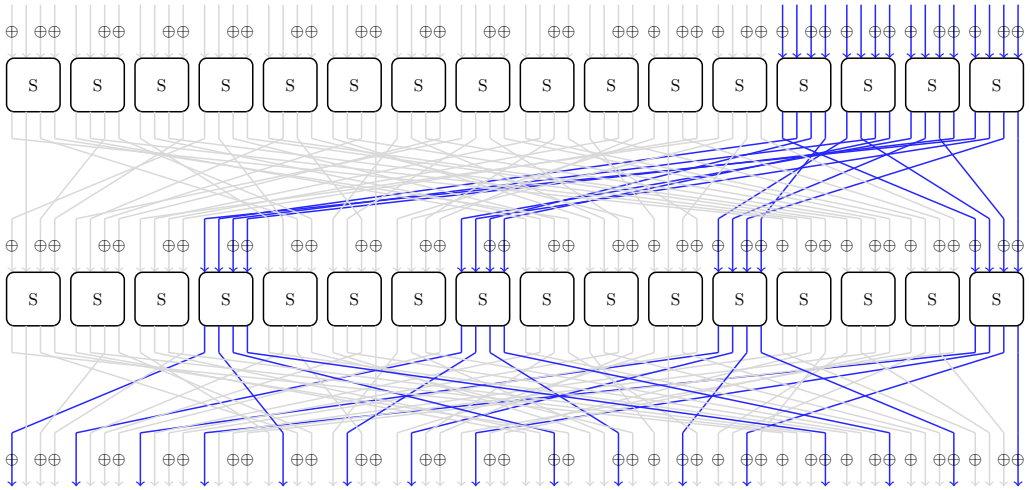


Figure 2: The Quotient-Remainder Group $\mathcal{QR}$-0 in the GIFT-64 permutation

selected for key derivation. If an attacker obtains the round keys of two consecutive rounds for GIFT-128 or four for GIFT-64, they can recover the master key. This property is utilized for both GIFT versions in later sections.

2.2.1 The $\mathcal{QR}$ Structures in GIFT Permutation

GIFT is an SPN structure whose permutation layer consists of bit permutation only. This can be grouped in 4 or 8 groups of 16-bit to 16-bit permutation structures depending upon the versions, called the $\mathcal{QR}$ structures (Refer Figure 2). This follows the convention that the output bits of 4 nibbles map to the input bits of 4 nibbles in the next round. Consider a nibble N_i^l with $i = 4 * a + b$. This belongs to the a-th quotient group in R^l and b-th remainder group in R^{l+1} for GIFT. The number of permutation group is different for the versions of GIFT and thus makes 4 $\mathcal{QR}$ groups for GIFT-64 and 8 groups for GIFT-128. From this convention, it can be seen that to recover the round keys of two consecutive rounds, the attacker does not have to guess the full round key of the cipher. She can guess the round keys for individual groups. The original round key remains in the Cartesian product of the $\mathcal{QR}$ group keys. We exploit this structuring technique in our attack to recover the round keys for the both versions of GIFT. We call the i-th $\mathcal{QR}$ group as $\mathcal{QR}_i$.

2.3 BAKSHEESH [BBC+23]

The BAKSHEESH block cipher is designed to optimize both hardware and software aspects of its implementation and has been claimed by its designers to be an improvement over GIFT. It consists of 35 rounds and operates on a state size of 128-bits. Each round of the cipher encompasses four operations: SBox-Layer, bit-permutation, addition of round constant, and addition round key. The linear layer of the cipher employs bit-permutation, which is the same as that used in GIFT-128. In the add round constants layer, a 6-bit round constant is added to specific positions within the state. Unlike GIFT-128, BAKSHEESH has key whitening layer at the beginning of the encryption function, which makes a total of 36 round keys for the cipher. The initial whitening key is same as the master key. Subsequent round keys are derived through a one-bit circular right shift of the previous round key, i.e., $k^{i+1} \leftarrow k^i \ggg 1$. The key schedule represents a notable difference between GIFT-128 and BAKSHEESH, which plays a crucial role in our attack. Additionally, another significant difference is how the round key is incorporated during the encryption process. Unlike

GIFT-128, in BAKSHEESH, the round key is added to the full state rather than specific bits, using a bitwise XOR operation.

It is worth noting that the key schedule process is invertible even with the knowledge of single round key. This invertibility of the key schedule mechanism in BAKSHEESH, is fully exploited in our attack.

3 ToFA: Impossible Differential Induced Fault Analysis

In this section we introduce the idea of ToFA starting with the notion of truncated differentials in the state of GIFT. It can be noted that in this part of the attack we primarily leverage the $\mathcal{QR}$ group structure. Due to the random byte fault model used in this work, we capture the state-wide input differences that are permissible though the following.

Definition 1 (Admissible Input Difference (AID))**.** An admissible input difference is a non-zero truncated difference whose Hamming-weight is at most 8 belonging to the following set.

$$\mathcal{S}_b^{\Delta} = \left\{ \left[*^8 0^{b-8}\right] \ggg k \mid 0 \leq k < b-8, *^8 \in \{0,1\}^8 \setminus \mathbf{0} \right\} \tag{1}$$

Here b denotes the state-size. For e.g., in case of GIFT-64, $\mathcal{S}_{64}^{\Delta}$ captures the set of all differences that can be induced by a random byte-fault in the intermediate state.

Definition 2 (Spread of an AID (SAID))**.** The Spread of an AID is the distance between the most significant non-zero bit and the least significant non-zero bit values. For e.g. $\mathsf{SAID}(0x0^{13} \cdot 1e \cdot 0 \in \mathcal{S}_{64}^{\Delta}) = 3$.

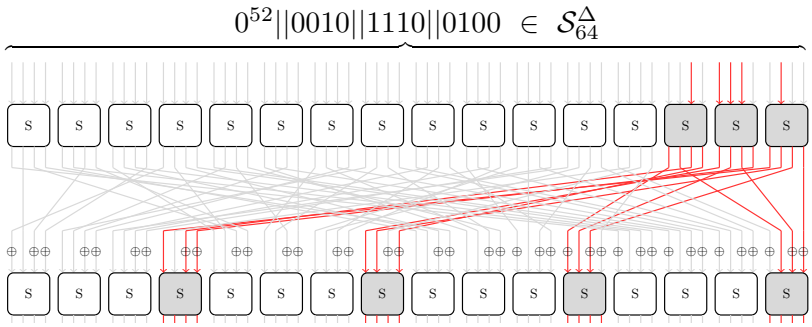
The notion of SAID is introduced for conveniently capturing the maximal fault diffusion. The maximum SAID for a random byte fault is 7 while minimum is zero corresponding to single bit being non-zero. Table 2 shows the maximum number of SBox-es and $\mathcal{QR}$-groups that can be activated by a random byte fault based on the SAID of the induced input difference.

Table 2: SAID relation with activity pattern considering maximum diffusion of the fault

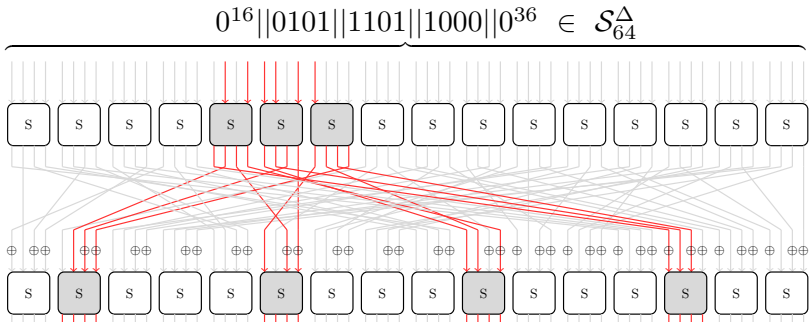
SAID	Max #Active SBox-es	Max #Active $\mathcal{QR}$-groups
0	1	1
1,2,3,4	2	2
5,6,7	3	2

Observation 1 (AID - $\mathcal{QR}$ Correlation)**.** *Any AID, $s \in \mathcal{S}_b^{\Delta}$ will diffuse following the $\mathcal{QR}$ group structure and can activate at most **two** $\mathcal{QR}$ groups.*

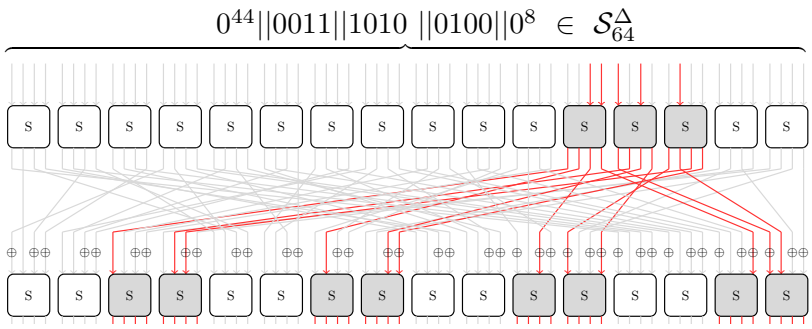
Remark 1. This observation is evident from Table 2. For any SAID > 0, there are at least two active bits. Now since max SAID is 7, these active bits will activate two SBox-es which can at max be separated by another SBox. Additional bits active in between the most and least significant active bits can activate the intermediate SBox as well leading to max of three SBox-es being active. Now, based on the location of the induced byte fault, the active SBox-es can all belong to the same $\mathcal{QR}$ group (referred to as Type-I correlation - Refer Figure 3a) or can be divided between at most two *adjacent* $\mathcal{QR}$ groups (referred to as Type-II correlation - Refer Figure 3c). In the remainder of the work, we always assume that the AID affects three consecutive SBox-es. It can be noted that any sub-case (i.e., if the induced fault affects two or one nibble) is captured through this assumption. From the *fault-model perspective* this can be visualized as a further relaxation of the random-byte model, where our fault model can *tolerate the injected fault to spread to up to any three contiguous nibbles.*



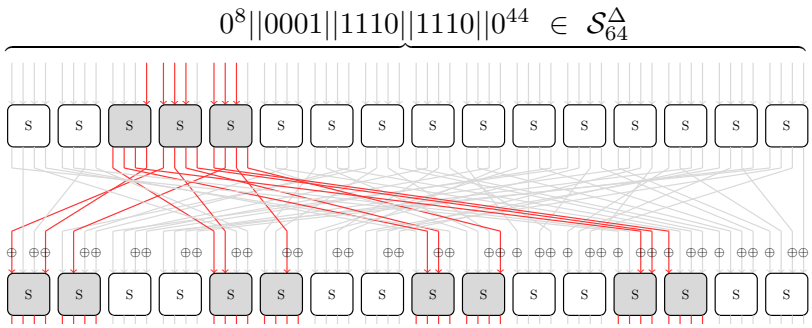
(a) Type-I correlation for $0^{54} *^8 0^2 \in \mathcal{S}_{64}^{\Delta}$



(b) Type-I correlation for $0^{17} *^8 0^{39} \in \mathcal{S}_{64}^{\Delta}$



(c) Type-II correlation for $0^{46} *^8 0^{10} \in \mathcal{S}_{64}^{\Delta}$



(d) Type-II correlation for $0^{11} *^8 0^{45} \in \mathcal{S}_{64}^{\Delta}$

Figure 3: Demonstration of AID - $\mathcal{QR}$ group correlation in GIFT-64

Observation 2 (One-Round Truncated Cluster)**.** *Based on the correlation induced by any AID in the r^{th} round, one can enumerate the cluster of permissible truncated difference patterns ($\mathcal{T}_b^{\Delta_1}$) after $(r+1)^{th}$ round using the predecessor set ($\mathcal{P}_b^{\Delta_1}$) which is represented as below for $b = 64$. For $b = 128$, please refer Appendix A.*

$$\mathcal{T}_{64}^{\Delta_1} = \left\{t \in \{0,1\}^{64} | t \preceq p\right\}, \text{ where}$$

$$p \in \mathcal{P}_{64}^{\Delta_1} = \left\{ \begin{array}{cc} \overbrace{\begin{array}{l}[000b\ 000d\ 000e\ 0007] \\ [0007\ 000b\ 000d\ 000e] \\ [00b0\ 00d0\ 00e0\ 0070] \\ [0070\ 00b0\ 00d0\ 00e0] \\ [0b00\ 0d00\ 0e00\ 0700] \\ [0700\ 0b00\ 0d00\ 0e00] \\ [b000\ d000\ e000\ 7000] \\ [7000\ b000\ d000\ e000]\end{array}}^{\text{Type-I}} & \overbrace{\begin{array}{l}[0086\ 0043\ 0029\ 001c] \\ [0094\ 00c2\ 0061\ 0038] \\ [0860\ 0430\ 0290\ 01c0] \\ [0940\ 0c20\ 0610\ 0380] \\ [8600\ 4300\ 2900\ 1c00] \\ [9400\ c200\ 6100\ 3800]\end{array}}^{\text{Type-II}} \end{array} \right\}$$

Remark 2. The above observation is completely dictated by the $\mathcal{QR}$ structures that are an inherent part of GIFT design. It has an interesting implication considering the entire cluster. Any one-round truncated pattern (t') which is not a predecessor of p i.e., $t' \not\preceq p$ is an *impossible* truncated differential. Alternately, for $s \in \mathcal{S}_{64}^{\Delta}$ and $t' \not\preceq p$ where $p \in \mathcal{P}_{64}^{\Delta_1}$, we have $\Pr[s \xrightarrow{\text{1-Round GIFT-64}} t'] = 0$.

Observation 3 (Two-Round Truncated Cluster)**.** *Observation 2 can easily be extended over two rounds so as to enumerate permissible truncated difference patterns after $(r+2)^{th}$ round ($\mathcal{T}_b^{\Delta_2}$) due to any AID at the r^{th} round using the corresponding predecessor set ($\mathcal{P}_b^{\Delta_2}$) given as below for $b = 64$. Again refer Appendix A for the predecessor set for state-size 128.*

$$\mathcal{P}_{64}^{\Delta_2} = \left\{ \begin{array}{cc} \overbrace{\begin{array}{l}[8888\ 4444\ 2222\ 1111] \\ [1111\ 8888\ 4444\ 2222] \\ [2222\ 1111\ 8888\ 4444] \\ [4444\ 2222\ 1111\ 8888]\end{array}}^{\text{Type-I}} & \overbrace{\begin{array}{l}[9999\ cccc\ 6666\ 3333] \\ [3333\ 9999\ cccc\ 6666] \\ [6666\ 3333\ 9999\ cccc]\end{array}}^{\text{Type-II}} \end{array} \right\}$$

Remark 3. This is also due to the two-round diffusion property of GIFT-64 and shows a way to achieve two-round impossible differential trails. In the subsequent round truncated differentials will fully diffuse in the state and will no longer be viable.

Observation 4 (Three-Round Truncated Cluster (GIFT-128, BAKSHEESH))**.** *This observation is only applicable to 128-bit state-size where the truncated differentials induced due to AID in r^{th} round can be propagated to the $(r+3)^{th}$ round to enumerate ($\mathcal{T}_{128}^{\Delta_3}$) based on ($\mathcal{P}_{128}^{\Delta_3}$).*

$$\mathcal{P}_{128}^{\Delta_3} = \left\{ \begin{array}{l} \texttt{[aaaa aaaa 5555 5555 aaaa aaaa 5555 5555]} \\ \texttt{[5555 5555 aaaa aaaa 5555 5555 aaaa aaaa]} \\ \texttt{[ffff ffff ffff ffff ffff ffff ffff ffff]} \end{array} \right\}$$

Remark 4. The last element of $\mathcal{P}_{128}^{\Delta_3}$ is interesting as it implies full diffusion. This is attributed to the fact that there exist a few AIDs $\in \mathcal{S}_{128}^{\Delta}$ (7 out of 120 to be precise) for which truncated differentials can only be propagated for two rounds. This further shows that if a random-byte fault induces those AIDs then three-round cluster will not be formed. However, the probability of such cases is less and hence we ignore this in the rest of the work. Based on the above three observations we have the following generalized impossibility.

$$\left(\forall t \in \{0,1\}^b | t \not\preceq p \in \mathcal{P}_b^{\Delta_r}\right) \text{ and } \left(\forall s \in \mathcal{S}_b^{\Delta}\right), \Pr\left[s \xrightarrow[\text{GIFT}-b]{r-\text{Round}} t\right] = 0, \quad \begin{cases} 1 \le r \le 2, b = 64 \\ 1 \le r \le 3, b = 128 \end{cases} \tag{2}$$

Equation 2 is the basis of the following property which leads us to build restricted truncated difference based fault invariants. Since BAKSHEESH borrows the bit-permutation from GIFT-128, all the above observations are applicable to it as well.

3.1 Truncated Differential Induced Fault Invariants

Property 1 (Two-Round Fault-Invariants (GIFT-64))**.** *Any random byte fault injected in the input of r^{th} round will result in a difference $\in \mathcal{S}_{64}^{\Delta}$, and after **two** rounds of GIFT-64 will result in a output difference $\in \mathcal{T}_{64}^{\Delta_2}$ that will satisfy Equation 2.*

For the sake of completeness, we also state the equivalent property pertaining to 128-bit block-size due to Equation 2.

Property 2 (Three-Round Fault-Invariants (GIFT-128/ BAKSHEESH))**.** *Any random byte fault injected in the input of r^{th} round will admit a difference $\in \mathcal{S}_{128}^{\Delta}$, and after **three** rounds of GIFT-128/BAKSHEESH will admit an output difference $\in \mathcal{T}_{128}^{\Delta_3}$ that will satisfy Equation 2.*

3.1.1 Exploiting Local Patterns with $\mathcal{QR}$ Structures

It is interesting to note that the invariants due to Property 1 and 2 need not be only exploitable globally. They can also be exploited locally which is due to the already beneficial $\mathcal{QR}$ structure and reduces the key-guessing complexity. For the rest of this paragraph refer to Figure 4. We consider the case of GIFT-128 which has Property 2. Random byte faults can be injected at the input of R^{35} and will induce truncated differential trails $\in \mathcal{T}_{128}^{\Delta_1}, \mathcal{T}_{128}^{\Delta_2}$ and $\mathcal{T}_{128}^{\Delta_3}$ at the input of R^{36}, R^{37} and R^{38} respectively. We need to partially guess the R^{38} and R^{39} round-keys but only pertaining to SBox-es belonging to the same $\mathcal{QR}$ group (different colors in R^{38} and R^{39} in Figure 4 depict different $\mathcal{QR}$ groups) to partially decrypt two-rounds. If the resultant bit patterns that emerge corresponding to the $\mathcal{QR}$ group of round $(r-1)$ do not conform to the corresponding part of $\mathcal{T}_{128}^{\Delta_3}$, we eliminate the key-values guessed since they violate Equation 2 and imply an impossibility. We repeat this step independently for each $\mathcal{QR}$ group. While this reduces key-spaces of individual $\mathcal{QR}$ groups, we also leverage their combinations and do so in a hierarchical way. First we take groups of two and use the Cartesian product of reduced key-spaces to decrypt and verify the sub-patterns. Next we combine groups of four and finally all the eight $\mathcal{QR}$ groups are used together to to eliminate the candidates for K^{39} and K^{38} exploiting non-conformance to $\mathcal{T}_{128}^{\Delta_3}$. At this stage, we have a reduced key-space for R^{39} and R^{38}. The strategy is same for BAKSHEESH while it adapts easily for GIFT-64.

3.1.2 Exploiting Multi-Round Impossibility

As stated earlier, truncated differentials develop over multiple rounds which can be exploited incrementally as we decrypt further rounds. In case of GIFT-128, owing to the key-schedule K^{39} and K^{38} are sufficient to invert the key-schedule and get candidates for K^{37} and K^{36} which can be eliminated using $\mathcal{T}_{128}^{\Delta_2}$ and $\mathcal{T}_{128}^{\Delta_1}$ respectively. For BAKSHEESH, the key-schedule is more simpler and only the last round-key is sufficient to invert it. Thus for BAKSHEESH relation between last and second-last key acts a secondary filter for key-elimination along with $\mathcal{T}_{128}^{\Delta_2}$ and $\mathcal{T}_{128}^{\Delta_1}$ where we exploit multi-round impossibility. Though GIFT-64 requires the last four round-keys to invert the key-schedule, impossible truncated differential patterns can still be leveraged to recover three of the last four round keys.

With the basic framework of ToFA in place, we are now ready to delve into cipher-specific instantiations of the attack. In the next section, we illustrate ToFA in further detail starting with the 128-bit ciphers targeted in this work.

4 ToFA on GIFT-128 and BAKSHEESH

In this section, we will explore the process of recovering the master-key for GIFT-128 and BAKSHEESH using ToFA. Our approach involves utilizing the fault induced 3−round truncated differential trail in the random byte fault model. The attacker is free to induce a random byte fault and induce and AID which is exploited as per properties and insights gained from the impossible differential trail observation discussed in Section 3.

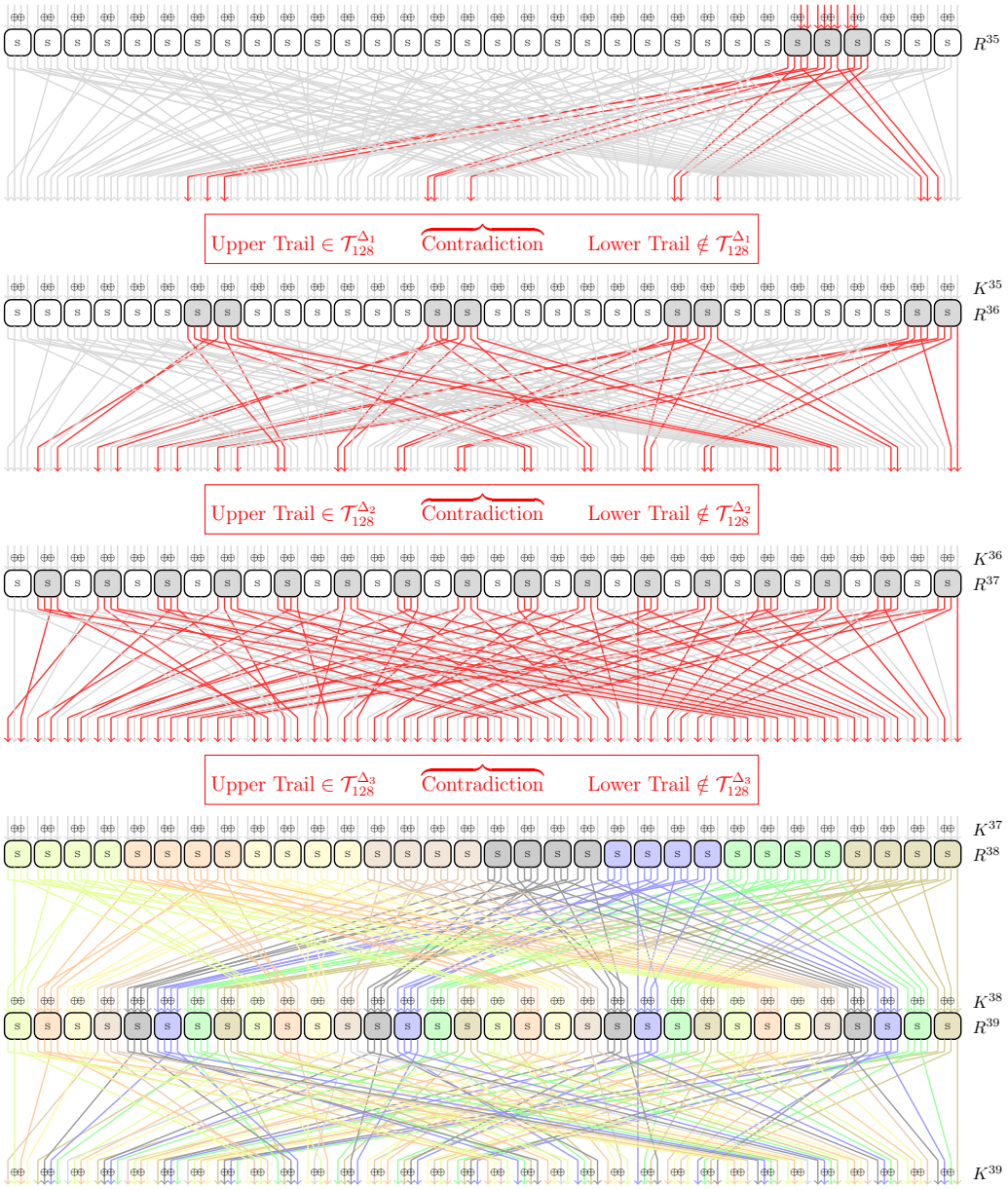


Figure 4: Multi-round contradiction that can be exploited in GIFT-128 due to 3-round truncated differential trail. The figure represents one instance of the fault injection in R^{35}. Here SAID $(0^{104}||0011||1111||1100||0^{12}) = 7$

Suppose the fault is given at any random byte of S^r at the r-th round. Based on Observation 1, the resulting difference propagates to a maximum of two remainder groups

(the specific groups depend on the fault location) of S^{r+1} in the next round. This propagation generates a cluster of permissible truncated difference patterns denoted as $\mathcal{T}_{128}^{\Delta_1}$, which is detailed in Observation 2. In the subsequent round, the difference further spreads to a maximum of 16 out of 32 nibbles in accordance with Observation 3, resulting in another cluster of permissible truncated difference patterns denoted as $\mathcal{T}_{128}^{\Delta_2}$. If we examine one more round, we observe that the outputs of these specific SBox-es are connected to precisely two inputs of each SBox of S^{r+3}. This connection generates a cluster of permissible truncated difference patterns denoted as $\mathcal{T}_{128}^{\Delta_3}$ (see Observation 4). We employ this three-round trail to launch an attack on GIFT-128 and BAKSHEESH. An overview of this trail is presented in Figure 4.

Next, we show the process of utilizing this trail to recover the master keys for both GIFT-128 and BAKSHEESH. The core idea, like any DFA, behind our attack lies in the introduction of random faults at arbitrary locations during the R^r round. By doing so, we aim to disrupt the encryption process and observe the effects of these faults on the resulting faulty ciphertext. After injecting the faults, we proceed to decrypt the corresponding fault-free ciphertext and faulty ciphertext pairs obtained from the output of R^{r+5}, by guessing the round keys. During this decryption phase, we analyze the decrypted ciphertext-faulty ciphertext pairs for specific patterns. If the decrypted pairs exhibit certain predetermined patterns that align with the expectations derived from the three-round trail we discussed earlier, it indicates that our guessed key might be correct. On the other hand, if the decrypted pairs do not conform to these expected patterns, we discard the guessed key. This elimination forms the basis of ToFA.

Algorithm 1 CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$

Input: A list of $(C, C') \in \mathcal{C}$, Key, A shift value s
Output: True/False
1: $(\mathsf{Key}_{38}, \mathsf{Key}_{39}) \leftarrow \mathsf{Key}$
2: **for** $(C, C') \in \mathcal{C}$ **do** ▷ Decrypt each pair of ciphertext through last two rounds
3: $D = SBox^{-1}(P^{-1}(SBox^{-1}(C \oplus \mathsf{Key}_{39}) \oplus \mathsf{Key}_{38} \oplus RC_{38}))$
4: $D' = SBox^{-1}(P^{-1}(SBox^{-1}(C' \oplus \mathsf{Key}_{39}) \oplus \mathsf{Key}_{38} \oplus RC_{38}))$
5: $\Delta \leftarrow D \oplus D'$
Check permissible truncated difference pattern conformance
6: **if** $(0^{(128-(2*s))}||\Delta_i||0^s) \notin \mathcal{T}_{128}^{\Delta_3}$ **then**
7: **return** False
8: **return** True

4.1 Master-Key Recovery GIFT-128

One may recall that GIFT-128 is an 128-bit state SPN block cipher with 40 rounds. We start our discussion with the recovery strategy for the round keys of the last two rounds (i.e., K^{39} and K^{38}) focusing on each $\mathcal{QR}$ group independently, as described in Subsubsection 3.1.1. The $\mathcal{QR}$ groups for the key-guesses are shown in Figure 4 with different colors. For example, the green color in R^{38} and R^{39} represents $\mathcal{QR}_1$. The fault is injected at the input of R^{35} and we get a pair (C, C') at the output of GIFT-128. It is important to note that during the encryption process, each nibble of the state is XOR-ed with only two key bits. As a result, there are 2^{16} possible keys for each $\mathcal{QR}$ group. So, to decrypt the ciphertexts through a specific $\mathcal{QR}$ group in R^{39} and R^{38}, we need to guess 16 key bits. This includes an 8-bit key required to decrypt the four SBox-es in the remainder group of R^{39}, as well as another 8-bit key needed to decrypt the four SBox-es in the corresponding quotient group of R^{38}. Therefore, for each $\mathcal{QR}$ group in rounds 39 and 38, we make a 16-bit key guess and decrypt each ciphertext pair (C, C') to obtain the 16 input bits of the quotient group

in R^{38} and compute the difference.

At this stage, we examine the obtained 16-bit difference and check if it conforms to any of the permissible truncated difference patterns ($\mathcal{T}_{128}^{\Delta_3}$). This verification step is performed in Step 4−9 of Algorithm Algorithm 2. In Step 6, we invoke the CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$ subroutine (Algorithm 1), which examines the truncated difference patterns after three rounds from the fault injection. From this distinguisher, we acquire a reduced candidate set of 16 key bits for each $\mathcal{QR}$ group in R^{39} and R^{38}. These sets are denoted as $\mathcal{L}_1^i$ for $i \in \{0, 1, ..., 7\}$. The Cartesian product of these sets, constitutes a set of candidate keys for R^{39} and R^{38}. However, if we combine all these 8 sets at once, the size of the Cartesian product set may go beyond practical limits. Instead, we take an incremental approach and combine these sets in three steps. At the Step 10 in Algorithm 2 we combine two consecutive $\mathcal{QR}$ groups. In Step 19, we combine four consecutive $\mathcal{QR}$ groups. Finally, at Step 28, we combine all 8 groups together.

By following these step-by-step combinations, we gain a significant advantage. At each step, we can utilize the CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$ subroutine to reduce the size of the key set. This reduction helps manage the computation and memory requirements and improves the efficiency of the algorithm. We report the experimental results on the average size of the remaining key space for various combinations of $\mathcal{QR}$ groups in Table 4. At the end of Algorithm 2, we get a set of $\mathsf{K}^{38}||\mathsf{K}^{39}$, representing the candidate values for the round keys K^{38} and K^{39}. With these last two round keys, we can invert GIFT-128 key-schedule due to is very design thereby recovering the master key.

Up until now, we have focused solely on utilizing the impossibilities implied by Observation 4 to reduce the key-space. However, we have observed that there is additional potential for further reduction. Using the reduced candidate set $\mathsf{K}^{38}||\mathsf{K}^{39}$ of last two round keys, we can reconstruct the round keys for the preceding two rounds, namely K^{37} and K^{36}. Using the key K^{37} we further decrypt each pair of ciphertexts to get input difference, Δ_{37} at the input of R^{37}. At this stage, we leverage Observation 3 to further reduce the key-space. For each candidate derived key, we verify that if $\Delta_{37} \in \mathcal{T}_{128}^{\Delta_2}$ (as stated in Observation 3). If the verification fails, we discard that particular key suggestion from with the candidate for K^{37} was derived. Similarly, to achieve additional reduction in the key-space, we utilize Observation 2 by computing Δ_{36} at the input of R^{36}. The entire process is outlined in Algorithm 3.

Time Complexity. In this section, we analyze the time complexity associated with the attack discussed previously. We assume that we are working with f {fault-free, faulty} ciphertext pairs during the execution of the attack. Initially, we focus on recovering the round keys R^{38} and R^{39} corresponding to the $\mathcal{QR}$ groups independently, as outlined in line 4 of Algorithm 2. Here we call the CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$ procedure in Algorithm 1 for all $\mathcal{QR}$ groups and all 2^{16} possible keys. Within the CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$ procedure in Algorithm 1, we perform a total of 16 SBox inversions for each $\mathcal{QR}$ group and each key guess. Specifically, this involves two $SBox^{-1}$ operations for both D and D' in steps 3 and 4 in Algorithm 1) for each $\mathcal{QR}$ group and each guess of the key. Consequently, this results in a complexity of $f \times (2^{16} \times 8 \times 16) = 2^{16} \times 2^7 \times f$ SBox inversions.

At this stage, we have effectively reduced the key space corresponding to each $\mathcal{QR}$ group independently. The extent of this reduction is directly proportional to the number of faults f. In Table 4, we report the reduced key-space size for various values of f. For simplicity, we denote the reduced key space as $\kappa_1(f)$ (denoted in the column of $g = 1$ in Table 4).

Next, we proceed to combine two $\mathcal{QR}$ groups, as illustrated in 12 of Algorithm 2. Since each $\mathcal{QR}$ group involves 16 $SBox^{-1}$ operations, the total number of $SBox^{-1}$ operations for decrypting two rounds within the CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$ procedure amounts to 4×32. Hence, the time complexity for this step can be expressed as $f \times (\kappa_1(f)^2 \times 4 \times 32)$.

Algorithm 2 KeyRecovery3839-GIFT-128

Input: A list of $(C, C') \in \mathcal{C}$
Output: A list of master keys $\mathcal{K}$
Invert Each pair $(C, C') \in \mathcal{C}$ of through last bit-permutation and round constant addition layer and save in the same list $\mathcal{C}$
Prepare eight lists $\mathcal{C}^i$ for $i \in \{0, 1, ..., 7\}$ of 16 bit values from $\mathcal{C}$ for each $\mathcal{QR}$ groups
Initialize eight lists $\mathcal{L}_1^i$ for $i \in \{0, 1, ..., 7\}$ for each $\mathcal{QR}$ groups
for $i = 0$ to 7 **do** ▷ For each $\mathcal{QR}$ group
    for $\mathsf{Key} = 0$ to $2^{16} - 1$ **do**
        if CheckPattern-$\mathcal{T}_{128}^{\Delta_3}(\mathcal{C}^i, \mathsf{Key}, 16 * i) = \mathsf{True}$ **then**
            $\mathcal{L}_1^i = \mathcal{L}_1^i \cup \{\mathsf{Key}\}$
        else
            Break
Combining Two Consecutive $\mathcal{QR}$ Groups
Prepare four lists $\mathcal{C}^i$ for $i \in \{0, 2, 4, 6\}$ of 32 bit values from $\mathcal{C}$ for each two consecutive $\mathcal{QR}$ groups
Initialize four lists $\mathcal{L}_2^i$ for $i \in \{0, 2, 4, 6\}$ for each two consecutive $\mathcal{QR}$ groups
for $i \in \{0, 2, 4, 6\}$ **do** ▷ For each pair of two consecutive $\mathcal{QR}$ groups
    for $(\mathsf{Key}_0, \mathsf{Key}_1) \in \mathcal{L}_1^i \times \mathcal{L}_1^{i+1}$ **do**
        $\mathsf{Key} \leftarrow \mathsf{Key}_0 || \mathsf{Key}_1$
        if CheckPattern-$\mathcal{T}_{128}^{\Delta_3}(\mathcal{C}^i, \mathsf{Key}, 16 * i) = \mathsf{True}$ **then**
            $\mathcal{L}_2^i = \mathcal{L}_2^i \cup \{\mathsf{Key}\}$
        else
            Break
Combining Four Consecutive $\mathcal{QR}$ Groups
Prepare two lists $\mathcal{C}^i$ for $i \in \{0, 4\}$ of 64 bit values from $\mathcal{C}$ for each four consecutive $\mathcal{QR}$ groups
Initialize two lists $\mathcal{L}_3^i$ for $i \in \{0, 4\}$ for each four consecutive $\mathcal{QR}$ groups
for $i \in \{0, 4\}$ **do** ▷ For each four consecutive $\mathcal{QR}$ groups
    for $(\mathsf{Key}_0, \mathsf{Key}_1) \in \mathcal{L}_2^i \times \mathcal{L}_2^{i+2}$ **do**
        $\mathsf{Key} \leftarrow \mathsf{Key}_0 || \mathsf{Key}_1$
        if CheckPattern-$\mathcal{T}_{128}^{\Delta_3}(\mathcal{C}^i, \mathsf{Key}, 16 * i) = \mathsf{True}$ **then**
            $\mathcal{L}_3^i = \mathcal{L}_3^i \cup \{\mathsf{Key}\}$
        else
            Break
Combining All $\mathcal{QR}$ Groups
Initialize a list $\mathcal{K}$ ▷ For full round keys
for $(\mathsf{Key}_0, \mathsf{Key}_1) \in \mathcal{L}_3^0 \times \mathcal{L}_3^4$ **do**
    if CheckPattern-$\mathcal{T}_{128}^{\Delta_3}(\mathcal{C}, \mathsf{Key}, 0) = \mathsf{True}$ **then**
        $\mathcal{K} = \mathcal{K} \cup \{\mathsf{Key}_0 || \mathsf{Key}_1\}$
return $\mathcal{K}$.

Algorithm 3 MasterKeyRecovery-GIFT-128

Input: List of $(C, C') \in \mathcal{L}_c$, Candidate list $\mathbb{K}$ of $K^{38}||K^{39}$
Output: A list of keys $\mathcal{K}$

for $K^{38}||K^{39} \in \mathbb{K}$ **do** ▷ For each key suggestion
  Prepare K^{37} and K^{36} from $K^{38}||K^{39}$
  Pass = True
  for $(C, C') \in \mathcal{L}_c^i$ **do** ▷ Decrypt each ciphertext pair's last three rounds
    $D = R^{-37}(R^{-38}(R^{-39}(C, K^{39}), K^{38}), K^{37})$
    $D' = R^{-37}(R^{-38}(R^{-39}(C', K^{39}), K^{38}), K^{37})$
    $\Delta_{37} \leftarrow D \oplus D'$
    if $\Delta_{37} \in \mathcal{T}_{128}^{\Delta_2}$ **then**
      $D = R^{-36}(D)$, $D' = R^{-36}(D')$
      $\Delta_{36} = D \oplus D'$
      if $\Delta_{36} \in \mathcal{T}_{128}^{\Delta_1}$ **then**
        Continue
      else
        Pass = False; Break
    else
      Pass = False; Break
  if Pass = True **then**
    Reverse key-schedule to get master key K from $K^{38}||K^{39}$
    $\mathcal{K} = \mathcal{K} \cup \{\mathsf{K}\}$
return $\mathcal{K}$

Following a similar approach, we can compute the time complexities for combining four $\mathcal{QR}$ groups and eight $\mathcal{QR}$ groups, which yield the expressions $f \times (\kappa_2(f)^2 \times 2 \times 64)$ and $f \times (\kappa_4(f)^2 \times 128)$, respectively (estimation of $\kappa_2(f)$ and $\kappa_4(f)$ are given in the columns of $g = 2$ and $g = 4$ in Table 4, respectively). Consequently, the theoretical estimation of the total time complexity Algorithm 2 can be summarized as the total number of $SBox^{-1}$ operations

$$(2^7 \times 2^{16} + 2^7\kappa_1(f)^2 + 2^7\kappa_2(f)^2 + 2^7\kappa_4(f)^2)f = (2^{16} + \kappa_1(f)^2 + \kappa_2(f)^2 + \kappa_4(f)^2)2^7 f.$$

Finally, to recover the master key, we utilize Algorithm 3, which applies the decryption oracle for three rounds over the reduced key space. The time complexity of Algorithm 3 is thus dominated by the previously discussed time complexity of Algorithm 2. The time requirements for practical implementations of the aforementioned attack are detailed in Table 5 for various values of f.

For the above attack, the key space can be reduced to unique by using only 3 {fault-free, faulty} ciphertext pairs, i.e., $f = 3$.

4.2 Master-Key Recovery BAKSHEESH

In this section, we illustrate ToFA on BAKSHEESH which has been proposed very recently as an improvement over GIFT. This attack constitutes a third-party fault based cryptanalysis of this lightweight 128-bit block-cipher with 35 rounds. The attack procedure for BAKSHEESH follows the same approach as we discussed for GIFT-128 in Subsection 4.1 and Algorithm 2. Since GIFT-128 and BAKSHEESH share the same bit-permutation, we can leverage all the impossible trails that we discussed for GIFT-128 in the context of BAKSHEESH as well. The fault is injected at round R^{30}, and we obtain a ciphertext pair (C, C') at the output of BAKSHEESH. Similar to GIFT-128, we recover the last two round keys

(i.e., K^{33} and K^{34}) for each $\mathcal{QR}$ group independently and combine them in three steps. However, BAKSHEESH provides additional avenue of key-space reduction due to its simple key-schedule as elaborated next.

4.2.1 Exploiting BAKSHEESH Key-Schedule as a Secondary Key-Filter

In the subsequent paragraphs we highlight two properties of BAKSHEESH primarily pertaining to its key-schedule that have kind of *orthogonal* effect on complexity of ToFA on BAKSHEESH.

Property 3 (Quadratic Increase in $\mathcal{QR}$ Key-Space for BAKSHEESH)**.** *The $\mathcal{QR}$ key-size for BAKSHEESH is double with regards to GIFT-128 leading to a cardinality of 2^{32} instead of 2^{16}*

Remark 5. BAKSHEESH is highly inspired by GIFT with some deviations, one of which is the decision to add round-keys to the full-state. The direct implication is the doubling of #key-bits added to individual $\mathcal{QR}$ groups which leads to the quadratic increase in the number of keys to be guessed for each $\mathcal{QR}$ group.

Property 4 (Rotational Relation in BAKSHEESH Adjacent Round-Keys)**.** *The i^{th} round-key (K^i) and the $(i+1)^{th}$ round-key (K^{i+1}) of BAKSHEESH are related as $\mathsf{K}^{i+1} \leftarrow \mathsf{K}^i \ggg 1$*

Remark 6. According to the designers of BAKSHEESH, a simple key-schedule was an important decision and another deviation from its predecessor GIFT. However, it also implies that recovering the last round-key leads to the master-key by simple left rotations.

As stated above Property 3 and Property 4 have an orthogonal effect on the key-elimination strategy adopted here implying that while Property 3 makes it difficult to mount ToFA, Property 4 makes it easier. Due to quadratic increase in key-guessing complexity due to Property 3, the impossible truncated differential filter ($\mathcal{T}_{128}^{\Delta_3}$) becomes less effective. This leads to reduced key-spaces of individual $\mathcal{QR}$ groups which are so large that the incremental combination strategy (as shown earlier to be effective for GIFT-128) fails due to combinations being prohibitively large in sizes. Here Property 4 comes to the rescue. We note that the relation between successive round-keys can act as a secondary filter. However, such a filter cannot be used in its entirety (due to impractical-size of last round-key after exploiting the primary impossibility filter) and thus we resort to the $\mathcal{QR}$ segregation which has been very beneficial until now in ToFA. The idea is to exploit key-bit relations between last two round-keys across $\mathcal{QR}$ groups. Since $\mathcal{QR}$ groups can be leveraged independently, this strategy leads to effective reduction. We explain this by making the following observation.

Observation 5 (Cross-$\mathcal{QR}$ Key-bit Relations between K^{i+1} and K^i)**.** *Some of the 16-bit key bits of any remainder group in K^{i+1} are equal to some of the 16-bit key bits of the corresponding quotient group in K^i implying that certain key bits (as outlined in Table 3) are shared between these two groups.*

Table 3: Number of common key-bits between $\mathcal{Q}$ and $\mathcal{R}$ groups in consecutive round keys for BAKSHEESH, considering different combinations of $\mathcal{QR}$ groups.

$\mathcal{QR}$ Group	0	1	2	3	4	5	6	7	01	23	45	67	0123	4567	01234567
#equal key-bits	3	3	3	3	3	3	3	3	8	8	8	8	32	32	128

Remark 7. For example, if we consider the $\mathcal{QR}_0$ group, we find that there are 3 key bits in the quotient group that are equal to some 3 key bits in the remainder group. This key-bit equality holds true for the other $\mathcal{QR}$ groups as well. Consequently, the effective size of the

key-space for each $\mathcal{QR}$ group becomes 29. Table 3 provides the number of equal key bits for various combinations of $\mathcal{QR}$ groups. This table offers insights into the strength of the key relations and the filtering effect achieved as more $\mathcal{QR}$ groups are combined.

Secondary Key Elimination Filter Leveraging Observation 5 It is now evident that in addition to checking the truncated difference patterns within $\mathcal{T}_{128}^{\Delta_3}$, we can also examine the key relations between two consecutive round keys. By considering these key relations, we can further constrain the key-space and reduce the number of potential candidate keys for each $\mathcal{QR}$ group. This additional constraint adds an extra layer of verification to the attack process. When combining two consecutive $\mathcal{QR}$ groups, we observe that there are 8 common key bits between the quotient group and the corresponding remainder group. Furthermore, when considering four consecutive $\mathcal{QR}$ groups together, the number of common bits increases to 32. In other words, as we progressively combine more $\mathcal{QR}$ groups, the key relations become stronger, providing an increasingly effective filtering mechanism. Thus, at each step, we further reduce the key-space by incorporating the constraints imposed by the impossible difference patterns and by checking the equality between two consecutive round keys. As the size of the reduced key space for various combination directly affects the memory and time requirement, we report the average size of the remaining key space for various combinations of $\mathcal{QR}$ groups in Table 4. This is further validated with real-world attack realization detailed in Section 6.

Ultimately, through these steps, we obtain a set of $\mathsf{K}^{33}||\mathsf{K}^{34}$, representing the candidate values for the round keys of R^{33} and R^{34}. By using the last round key K^{34}, we can reverse the key-schedule process and successfully recover the master key of BAKSHEESH. With a set of master key in hand, we can proceed to construct the round keys for the previous two rounds, namely K^{31} and K^{32}. Using the key K^{32}, we decrypt each pair of ciphertexts to obtain the input difference, Δ_{32}, at the input of R^{32}. At this point, if required, we utilize Observation 3 to further reduce the key-space by verifying if $\Delta_{32} \in \mathcal{T}_{128}^{\Delta_2}$. To achieve additional reduction in the key-space, we can also apply Observation 2 by computing Δ_{31} at the input of R^{31}.

Time Complexity Here the same procedure of Algorithm 2 is followed to reduce the key space of BAKSHEESH. This cipher uses a 128-bit round key in each intermediate round. Total 32 key bits are xored with the state in each $\mathcal{QR}$ group. Suppose f is the number of {fault-free, faulty} ciphertext pairs needed for this attack. Like GIFT-128, in a single $\mathcal{QR}$ group, 16 $SBox^{-1}$ are needed to perform CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$ procedure for each faulty fault-free pair. Hence the complexity to reduce each $\mathcal{QR}$ group becomes $(f \times 2^{32} \times 8 \times 16)$ $= (f \times 2^{32} \times 2^7)$.

Here also we combine multiple consecutive $\mathcal{QR}$ groups and use CheckPattern-$\mathcal{T}_{128}^{\Delta_3}$ in the intermediate round. We also have some extra filtering steps using the key bit relations of the cipher (given in Table 3). Note that, if $\kappa_g(f)$ for $g \in \{1, 2, 4\}$ is the reduced key space size after combining g consecutive $\mathcal{QR}$ groups, then the expression of the time complexity remains the same as GIFT-128 for the combination of $\mathcal{QR}$ groups. Hence the total time complexity can be expressed as the following:

$$(2^7 \times 2^{32} + 2^7\kappa_1(f)^2 + 2^7\kappa_2(f)^2 + 2^7\kappa_4(f)^2)f = (2^{32} + \kappa_1(f)^2 + \kappa_2(f)^2 + \kappa_4(f)^2)2^7 f.$$

The estimated values of $\kappa_g(f)$ for $g \in \{1, 2, 4\}$ can be found in Table 4. We find the best result for BAKSHEESH using 4 faults.

Table 4: Simulation result of the key space reduction for GIFT-128 and BAKSHEESH. The result shows the values of κ_g for g combinations of $\mathcal{QR}$ groups where $g = 1, 2, 4$ and 8. T = Time complexity in terms of $SBox^{-1}$.

Primitive	# Faults	FI point	Reduced key-space				T
			$g=1$	$g=2$	$g=4$	$g=8$	
GIFT-128	4	35	2^3	2^5	2^6	1	$2^{25.11}$
	3		2^5	2^8	2^{11}	1	$2^{30.63}$
	2		2^7	2^{12}	2^{18}	2^3	2^{44}
BAKSHEESH*	8	30	2^6	2^9	2^{11}	1	2^{42}
	6		2^7	2^{11}	2^{12}	1	$2^{41.59}$
	5		2^9	2^{13}	2^{16}	1	$2^{42.33}$
	4		2^{12}	2^{16}	2^{19}	1	$2^{47.04}$

⋆ The numbers reported for BAKSHEESH consider both difference patterns and key relations.

5 ToFA on GIFT-64

To perform the ToFA attack on GIFT-64 (a block cipher with a 64-bit state and 28 rounds), we utilize a 2−round impossible trail within GIFT-64 and execute the attack in a random byte fault model. Assuming a fault occurs at a randomly chosen byte of S^r during the r-th round, according to Observation 1, the resulting difference spreads to at most two remainder groups of S^{r+1} in the subsequent round. This propagation generates a set of valid truncated difference patterns denoted as $\mathcal{T}_{64}^{\Delta_1}$ (Observation 2) which leads to $\mathcal{T}_{64}^{\Delta_2}$ (Observation 3) in the next round. We employ this two-round trail as a distinguisher to recover the master key of GIFT-64. An overview of this trail is presented in Figure 5.

To recover the master key, we begin by first retrieving the last two round keys, K^{27} and K^{26} and like GIFT-128, we divide-and-conquer using each $\mathcal{QR}$ group independently. Later, we discuss the recovery of the third-to-last round key, K^{25}. The $\mathcal{QR}$ groups for the key-guesses are shown in Figure 5 with different colors (for instance green represents $\mathcal{QR}_0$). The fault is injected at the input of R^{24} and we get a pair (C, C') at the output of GIFT-64. Similar to GIFT-128, during the encryption process, each nibble of the state is XOR-ed with only two key bits. As a result, there are 2^{16} possible keys for each $\mathcal{QR}$ group. So, for each $\mathcal{QR}$ group in rounds 27 and 26, we make a 16-bit key guess and decrypt each ciphertext pair (C, C') to obtain the 16 input bits of the quotient group in R^{26} and compute the difference. The difference is then checked for conformance to any of the permissible truncated difference patterns from 2-round trails ($\mathcal{T}_{64}^{\Delta_2}$). From this distinguisher, we acquire a candidate set ($\mathcal{L}_1^i$ for $i \in \{0, 1, 2, 3\}$) of 16 key bits for each $\mathcal{QR}$ group in R^{26} and R^{27}. To combine these sets together, we use similar method as we consider for GIFT-128 and BAKSHEESH, i.e., following step-by-step process and use the distinguisher from 2-round trail at each step as shown in the CheckPattern-$\mathcal{T}_{64}^{\Delta_2}$ (Algorithm 4) subroutine to reduce the size of the candidate key set. The complete procedure is captured by Algorithm 5 at the end of which we get a set of $\mathsf{K}^{26}||\mathsf{K}^{27}$, representing the candidate values for the round keys of R^{26} and R^{27}.

Recovering K^{25} Previously we discussed how to recover last two rounds key (i.e., K^{27} and K^{26}) of GIFT-64 leveraging impossible truncated difference patterns. However, at this stage reduced key-space for the master key is still out of practically verifiable range. Hence to recover K^{25} without increasing number of faults we guess the round keys of K^{25} by the same fashion using the $\mathcal{QR}$ groups and use the truncated difference pattern for one round (given in $\mathcal{T}_{64}^{\Delta_1}$). For each suggestion for K^{27} and K^{26} form the output of Algorithm 5, we recover a set of possible keys for the 25-th round. To do this, again we guess 8-bit key

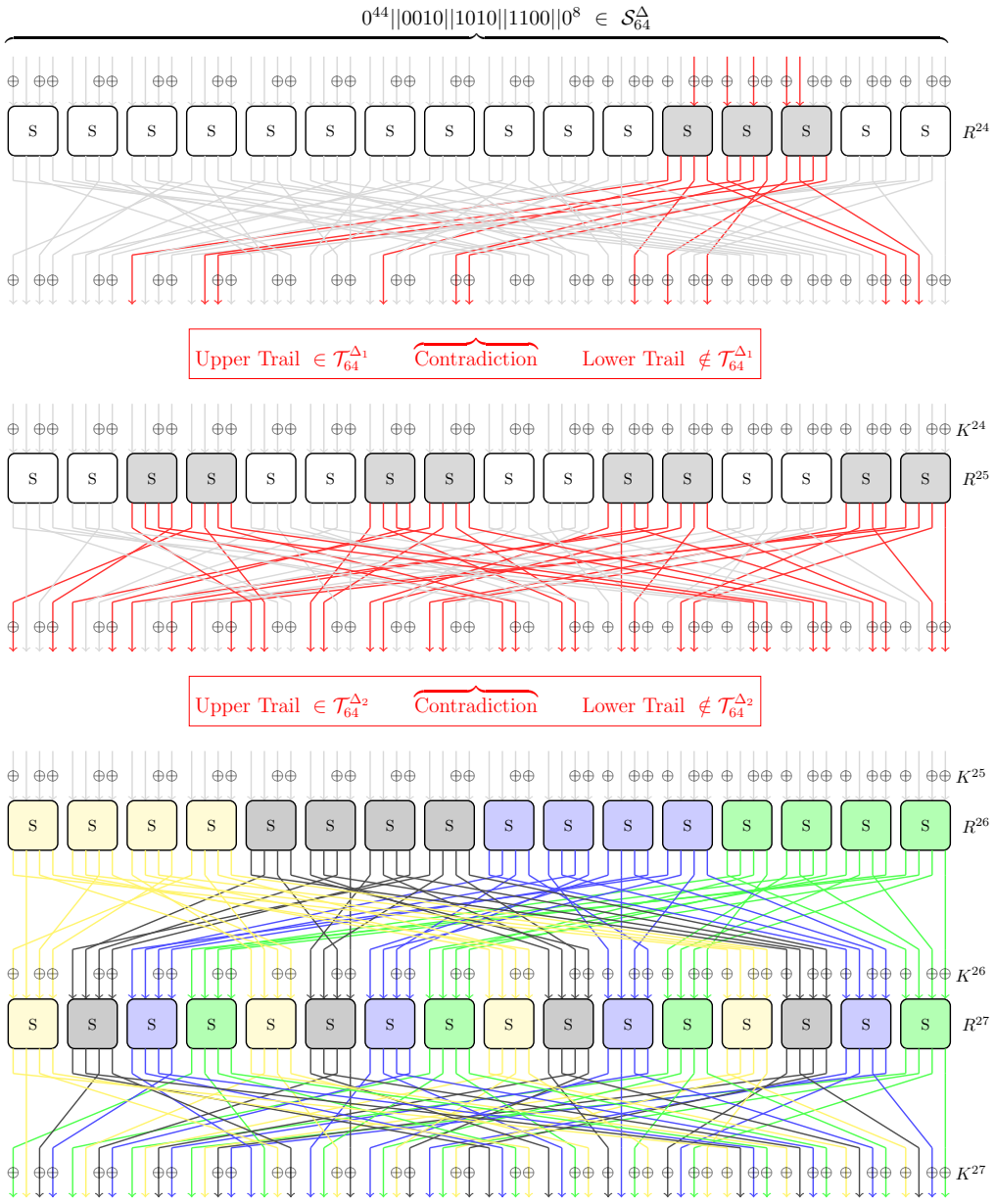


Figure 5: A conforming differential due to fault injection in $\mathcal{QR}$ group SBox in R^{24} of GIFT-64. For this case, SAID value is 7

Algorithm 4 CHECKPATTERN-$\mathcal{T}_{64}^{\Delta_2}$

Input: A list of $(C, C') \in \mathcal{C}$, Key, A shift value s
Output: True/False
$(\mathsf{Key}_{26}, \mathsf{Key}_{27}) \leftarrow \mathsf{Key}$
for $(C, C') \in \mathcal{C}$ **do** ▷ Decrypt each pair of ciphertext through last two rounds
  $D = SBox^{-1}(P^{-1}(SBox^{-1}(C \oplus \mathsf{Key}_{27}) \oplus \mathsf{Key}_{26} \oplus RC_{26}))$
  $D' = SBox^{-1}(P^{-1}(SBox^{-1}(C' \oplus \mathsf{Key}_{27}) \oplus \mathsf{Key}_{26} \oplus RC_{26}))$
  $\Delta \leftarrow D \oplus D'$
  Check permissible truncated difference pattern conformance
  if $(0^{(64-(2*s))}||\Delta_i||0^s) \notin \mathcal{T}_{64}^{\Delta_2}$ **then**
    return False
return True

Algorithm 5 KEYRECOVERY2627-GIFT-64

Input: A list of $(C, C') \in \mathcal{C}$
Output: A list of master keys $\mathcal{K}$
Invert Each pair $(C, C') \in \mathcal{C}$ of through last bit-permutation and round constant addition layer and save in the same list $\mathcal{C}$
Prepare eight lists $\mathcal{C}^i$ for $i \in \{0, 1, 2, 3\}$ of 16 bit values from $\mathcal{C}$ for each $\mathcal{QR}$ groups
Initialize eight lists $\mathcal{L}_1^i$ for $i \in \{0, 1, 2, 3\}$ for each $\mathcal{QR}$ groups
for $i = 0$ to 3 **do** ▷ For each $\mathcal{QR}$ group
  for Key $= 0$ to $2^{16} - 1$ **do**
    if CHECKPATTERN-$\mathcal{T}_{64}^{\Delta_2}(\mathcal{C}^i$, Key, $16 * i) =$ True **then**
      $\mathcal{L}_1^i = \mathcal{L}_1^i \cup \{\mathsf{Key}\}$
    else
      Break
  Combining Two Consecutive $\mathcal{QR}$ Groups
Prepare four lists $\mathcal{C}^i$ for $i \in \{0, 2\}$ of 32 bit values from $\mathcal{C}$ for each two consecutive $\mathcal{QR}$ groups
Initialize two lists $\mathcal{L}_2^i$ for $i \in \{0, 2\}$ for each two consecutive $\mathcal{QR}$ groups
for $i \in \{0, 2\}$ **do** ▷ For each pair of two consecutive $\mathcal{QR}$ groups
  for $(\mathsf{Key}_0, \mathsf{Key}_1) \in \mathcal{L}_1^i \times \mathcal{L}_1^{i+1}$ **do**
    Key $\leftarrow \mathsf{Key}_0||\mathsf{Key}_1$
    if CHECKPATTERN-$\mathcal{T}_{64}^{\Delta_2}(\mathcal{C}^i$, Key, $16 * i) =$ True **then**
      $\mathcal{L}_2^i = \mathcal{L}_2^i \cup \{\mathsf{Key}\}$
    else
      Break
  Combining All $\mathcal{QR}$ Groups
Initialize a list $\mathcal{K}$ ▷ For full round keys
for $(\mathsf{Key}_0, \mathsf{Key}_1) \in \mathcal{L}_2^0 \times \mathcal{L}_2^2$ **do**
  if CHECKPATTERN-$\mathcal{T}_{64}^{\Delta_2}(\mathcal{C}$, Key, $0) =$ True **then**
    $\mathcal{K} = \mathcal{K} \cup \{\mathsf{Key}_0||\mathsf{Key}_1\}$
return $\mathcal{K}$.

corresponding to each remainder group of R^{25} and decrypt one more round to get the input state R^{25}. To filter this 8-bit key, we check whether the input difference of R^{25} conforms to $\mathcal{T}_{64}^{\Delta_1}$ or not and use this as a key-guess elimination filter. This filter suggests a set of 8 key bits for each $\mathcal{QR}$ group in R^{25}. Again, we combine the key suggestion incrementally like before. At first we combine two consecutive $\mathcal{QR}$ and check if the combined input difference at R^{25} is in $\mathcal{T}_{64}^{\Delta_1}$. Then we combine all four groups together and check the same

pattern. The resulting process suggests a set of $\mathsf{K}^{25}||\mathsf{K}^{26}||\mathsf{K}^{27}$ representing the candidate values for the round keys of R^{25}, R^{26} and R^{27}.

Recovering Master Key From the key-schedule algorithm (see [BPP+17]) we can observe that to reverse the key-schedule algorithm and recover the master key, it is necessary to have 4 consecutive round-keys. As we have a set of keys for the last three rounds, we can recover the master key using exhaustive search for rest 32-bits. Thus if the size of the set of candidate $\mathsf{K}^{27}||\mathsf{K}^{26}||\mathsf{K}^{25}$ keys is t, our attack recover the master key for GIFT-64 with complexity $t \times 2^{32}$.

Time Complexity Here to reduce the master key space to 2^{32}, suppose f {faulty, fault-free} ciphertext pairs are needed. The procedure to recover the keys of R^{26} and R^{27} is given in Algorithm 5. As described in Section 4.1, here we first reduce the keyspace for 4 distinct $\mathcal{QR}$ groups, then combine the key spaces and use CheckPattern-$\mathcal{T}_{64}^{\Delta_2}$ to reduce the combined key space of the cipher. The effort to reduce the keyspace for each $\mathcal{QR}$ group involves $(f\times\ 4 \times 2^{16} \times 16)$ $SBox^{-1}$ operations. This reduces each $\mathcal{L}_1^i$ to 1 for each $\mathcal{QR}$ group i. Hence, unique key is recovered for the last two rounds. Since, unique key is recovered for every $\mathcal{QR}$ group, no further combinations are necessary. Now to recover K^{25}, first fully invert 2 rounds and then partially invert each of the $\mathcal{QR}$ group of the cipher. The size of the reduced key space lists after combining 2 and 4 $\mathcal{QR}$ groups becomes $2^{4.90}$ and 1 respectively. After recovering full 3 round keys also, 2^{32} possible keys remain for K^{24}. Hence the total time complexity becomes dominated by 2^{32} for GIFT-64. We find the best results using $f = 22$ for this case.

6 Implementing ToFA: Attack Realization

This section details the practical experimentation executed to realize the ToFA attack in real-world scenarios. To demonstrate, we utilized the 8-bit micro-controller ATXmega128D4-AU, which is resource constrained and supports light-weight encryption algorithms like tinyAES[2]. For fault injection we use the ChipWhisperer Lite (CW1173)[3] board where this microcontroller is set as the target (XMEGA). A clock-glitch as shown in Figure 6 is induced, resulting in some instructions getting skipped. For example, in a look-up-table based implementation of sub-bytes operation, the look up instruction or the store instructions can get skipped. This leads to the necessary faulty computation. The source codes for all of our experiments are available at the following repository[4].

6.1 Simulated Key-Recovery for Validation

Before delving into real-world attack realization, we conducted simulation experiments to ensure that post realization the key-recovery works as expected. The simulations are conducted by introducing a fault at any random byte with a random difference for both the primitives GIFT and BAKSHEESH. Using the strategy given in Subsection 4.1, we obtain a unique master key with only 3 random byte faults and reduce the key-space to 8 using only 2 faults for the case of GIFT-128. For the case of BAKSHEESH, we were able to obtain a unique master key using only 4 random byte faults. Interestingly, for BAKSHEESH, in most cases, we observed that the unique master key could be recovered even without utilizing Observation 3 and Observation 2. These findings underscore the

[2] https://github.com/newaetech/chipwhisperer/tree/develop/hardware/victims/firmware/crypto/tiny-AES128-C

[3] https://rtfm.newae.com/Targets/CW303%20XMEGA/

[4] https://github.com/ShibamCrS/Tofa.git

effectiveness of our attack methodology and highlight that the key-schedule mechanism in BAKSHEESH forms a prime exploitation point for ToFA on BAKSHEESH.

For the case of GIFT-64, we recover the unique key for the last two rounds ($\mathsf{K}^{27}||\mathsf{K}^{26}$) using only 8 random byte faults. For the last three rounds key ($\mathsf{K}^{25}||\ \mathsf{K}^{26}||\mathsf{K}^{27}$), our experiments show that on average 22 random byte faults are required to uniquely recover the last three rounds key and with 15 faults we can reduce the key-space to 2^9. These simulation results, although different from the actual attack result (discussed next), provide good upper and lower bounds on the number of faults required. The difference in the number of required faults between simulation and actual experimentation is due to the fact that 79% of the induced faults diffuse to only one nibble (as discussed in Subsection 6.3).

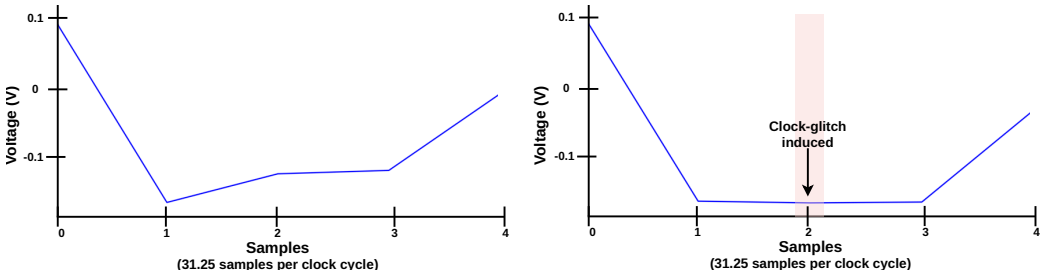


Figure 6: This diagram illustrates the voltage consumption under two conditions: (a) without clock glitch and (b) with clock glitch.

6.2 Results on GIFT-128

To program the microcontroller for running GIFT-128 encryption, we transformed its 8-bit word-size C-implementation into a '.hex' file. A trigger was then set to high and low before and after the desired round computations, respectively. Given that GIFT-128 is a constant-time implementation, the fault can always be induced at an estimated offset, thereby enabling us to inject the fault as discussed above. We induced the fault around the beginning of the 35th round, as shown in Figure 7. Since the first operation is sub-bytes, there is a high probability that most faults are induced due to instruction skip around this operation thereby affecting a single 4-bit word. The values of these faults vary, but since ToFA is invariant to them, we can achieve full key-recovery with just three faults in the best case. This demonstrates the feasibility of ToFA in practical settings.

The faults introduced result in faulty ciphertexts, which we utilized to uniquely recover the secret key by obtaining multiple such faulty and fault free ciphertext pairs. It is worth noting that with just two faults, the key space is already significantly reduced to only four potential keys on average. This trend continues with even fewer faults. Following this practical fault demonstration, we will now discuss other crucial factors, such as the attack range and success probability.

6.3 Fault Profiling of the ATXmega128D4-AU Micro-controller

Prior to discussing this in detail, we would like to reiterate that our attack imposes no restrictions on the Hamming-weight ($\mathcal{HW}$) of the fault beyond the requirement that it must be non-zero (otherwise it is not a valid fault). The theoretical framework enables that the fault can occur anywhere in the state and requires that it affects a maximum of 3 consecutive nibbles for GIFT-128 to mount a key-recovery attack. In terms of an actual fault, this translates to fault extendability to three consecutive computations. Notably our experimentation revealed that the impact of clock-glitches typically propagates to only one

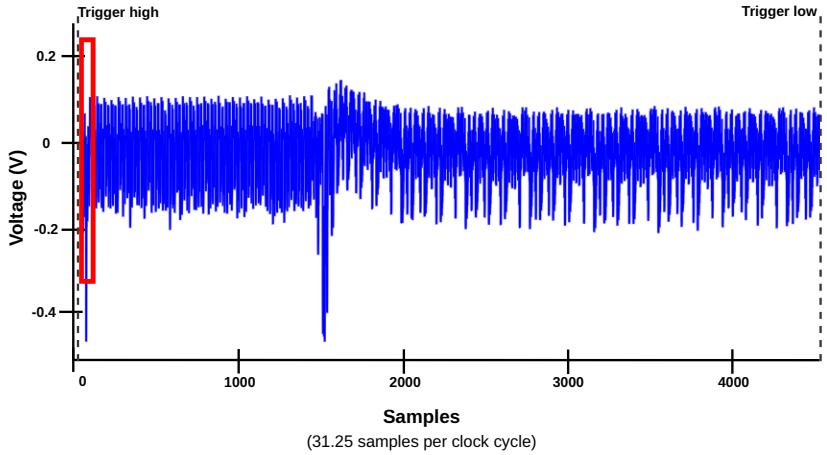


Figure 7: The diagram displays the faults introduced right at the onset of the 35th round, where the sub-bytes operation is anticipated to start. This shows a distinct pattern for every round, and if needed for entire encryption traces, this pattern can be used to partition the rounds.

computation. Hence, the success probability of achieving the desired faults is significantly high (79.5%).

The time-window for fault injection is 1.25% of the total GIFT-128 encryption runtime. This work relies on a random byte fault model which means that we do not need to know the value or spatial location of the fault. However, to do a feasibility analysis we injected random byte faults in the final round of GIFT-128 and from the ciphertext difference found that faults conforming to our attack setting can be achieved with a 79.5% success rate. The experimentation was carried out on 100 sample runs over the entire available attack range. This was accomplished by introducing clock glitches during the first 1000 samples of the last round. This analysis is detailed in Figure 8.

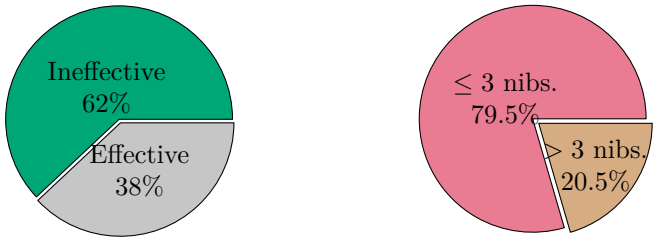


Figure 8: A pie chart representing the distribution of faults. The first chart shows that within the favorable time-window of fault injection, only 38% of the faults injected lead to faulty results. The second chart shows that close to 80% of the injected faults affect a maximum of 3 nibbles (nibs.) and are thus usable to mount ToFA.

As mentioned earlier, in the best case three faults is sufficient to perform full key-recovery, however, this is probabilistic event, and to be 100% sure and attacker would want to induce 12 faults, as shown in Figure 9. A similar study is also carried out for GIFT-64, which results in full key recovery with 20 faults. Finally, note that although in our experimentation we employ clock glitches, voltage glitches can also be used to induce faults. It is worth noting that precise faults, such as bit flips within the state, can also be mounted on FPGA implementations.

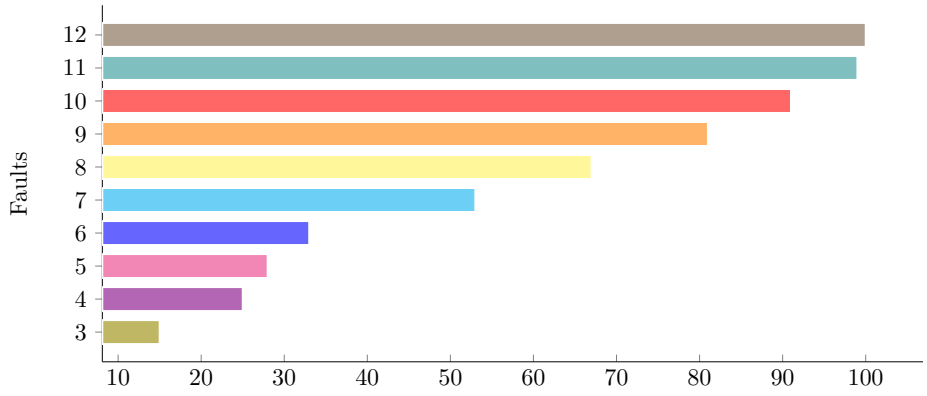


Figure 9: A Bar chart representing the percentage of unique key recoveries possible for a given number of faulty computations averaged over across 100 iterations. Relaxing the unique key recovery requirement significantly increases success rates. Even with a larger reduced key space, the correct key is always included, confirming the attack's correctness, and the unique key can be retrieved with minimal post-processing.

Countermeasure Analysis

A conventional countermeasure for protecting against fault attacks is redundancy. In ToFA we extend the attack up to 5 rounds, however, lower rounds are also vulnerable to fault analysis, without loss of generality. Therefore, we have added this countermeasure and tested that every fault is detected and results in an all zero ciphertext for GIFT-128. This protects the design against ToFA with 12.5% extra runtime overhead.

Another effective countermeasure against ToFA would be utilizing bit-sliced implementation technique. Such an implementation will require very precise faults in the registers containing data. Such registers are generally at the very least 32 to 64-bit large and cannot be processed using the 8-bit constraint microcontrollers, such the one utilized to mount the attack (ATXmega128D4-AU).

To enhance fault detection, we employ a duplication-based countermeasure starting from round 35. An additional state is created, and the previous state is copied to it. Both states undergo identical operations, ensuring that the final results match. At the end of the computation, an equality check is performed. If a fault occurs, this check will fail, effectively detecting potential attacks. While this approach is straightforward, it can be further strengthened by incorporating additional checks or XOR-based techniques to make fault injection more challenging for the attacker. Apart from simple duplication, infection-based countermeasures [GSSC15] can also be utilized in conjugation with redundancy-based countermeasure.

6.4 Software Implementation of The Key-recovery

Here, we present an in-depth analysis of the time requirements for the key-recovery, implemented in C++. We conducted all the experiments on an AMD EPYC 9554P machine, featuring a 3.7 GHz CPU and a 64-core processor. Given the parallelizable nature of the attack, we designed the implementation to run as a highly parallelized program, allowing us to maximize the throughput of the machine's multiple cores. The practical time requirements for various instances of the attack, under different number of faults, are provided in Table 5. The reported times represent the average execution time over 50 experiments, each with random master key and plaintexts. This table illustrates the effect of varying the number of faults on the total execution time.

Table 5: Attack implementation results.

Primitive	# Faults	Avg. Attack Time	Reduced Key-space
GIFT-128	8	2 seconds	1
	4	15 seconds	1
	3	58 seconds	1
	2	29.5 minutes	4
BAKSHEESH	8	3.5 minutes	1
	6	3.7 minutes	1
	5	7.5 minutes	1
	4	40 minutes	1
GIFT-64	22	43 milliseconds	1
	20	60 milliseconds	16
	16	63 milliseconds	256
	15	75 milliseconds	512

7 Discussion

This work provides insights in how classical cryptanalysis strategies can be exploited for physical attacks and reiterates the role of classical tools in making real-world attacks more feasible. A key contribution of the current work is complete/practical key-recovery attacks using a single point of fault injection owing to the independence from the round peeling strategy which most of the current fault attacks on GIFT rely on.

The ability to work in the random byte fault model makes ToFA an attractive proposition and helps it in standing out of all contemporary attacks which exploit the more restricted random nibble fault model. The exploitation of multi-round impossibility induced by the truncated differentials and hierarchical exploitation of $\mathcal{QR}$ group structures showcases full usage of all cipher properties of GIFT-64, GIFT-128 and BAKSHEESH. Moreover, for BAKSHEESH we observe that two separate design deviations of BAKSHEESH from GIFT-128 have an orthogonal effect on effectiveness of ToFA. In the design document of BAKSHEESH, the designers claim no security against DFA and attribute it to the choice of SBox that has 1-LS. However, in our analysis we find that it is the choice of the key-schedule that serves as the most effective key-elimination filter in ToFA.

In the introduction, we highlight that DEFAULT [BBB+21] fits the description of a GIFT-like cipher and could have been of a potential target of this current work. The designers of DEFAULT claimed that it is fault resistant by-design owing to the linear structures in the SBox. However, there have been a couple of interesting differential faults attacks [NDE22, JKP24] on DEFAULT which both exploit the LS in two different settings. In [NDE22], authors leverage multi-round linear sub-key bit-relations arising out of the LS using random nibble faults induced in third-last round of encryption. On the other hand [JKP24] exploits 5-round deterministic trails arising due to random *but known* bit faults induced during encryption. Our research reveals that LS SBox approach resists key-space reduction strategy introduced in this work provided that we are not able to exploit the sub-key relations. This is the reason that our attack is successful on BAKSHEESH (sub-keys related by one circular shift) but struggles on DEFAULT (sub-keys repeat after every 4-rounds). One workaround could be combing our strategy with the LS property of the SBox. However, we believe that this would entail a significant effort and can be taken up as an independent work.

8 Conclusion

This work introduces and practically verifies ToFA a fresh attempt at applying the idea of truncated impossible differentials to mount fault attacks on the micro-controller implementations of the popular lightweight block cipher GIFT and its recent successor BAKSHEESH using the widely used random byte fault model. This is the first FA on GIFT that recovers the key-space to practical limits (to the level of unique key-recovery) without *peeling* the cipher and is practically verified. So faults need to be induced at a fixed round only. Even in terms of fault penetration, ToFA outperforms all contemporary FA on GIFT variants. All this is achieved based on multiple observations due to truncated differential patterns augmented by $\mathcal{QR}$ group structures.

The observations translate to fault invariants which can be leveraged across multiple rounds for key-elimination. ToFA on GIFT-128 retrieves the unique master-key with only three random byte faults. On GIFT-64, ToFA gives the maximum theoretically possible reduction of 2^{32} with 22 random byte faults. This work also reports the first FA on the block cipher BAKSHEESH and recovers the unique key using four faults. Overall, ToFA constitutes the best FA on BAKSHEESH and GIFT, uses the widely used random byte fault model while giving new insights into FA vulnerability assessment of bit-oriented SPN block ciphers that leverages all facets of their design.

Acknowledgement

Shibam Ghosh was supported by the Israeli Science Foundation through grants No. 880/18 and 3380/19. Aikata was supported by the State Government of Styria, Austria - Department Zukunftsfonds Steiermark. Dhiman Saha was supported by the R&D in IT Division, Ministry of Electronics and Information Technology (MeitY), Government of India through sanction No. 4(4)/2022-ITEA dated 7th December 2023.

References

[AKS20] Aikata, Banashri Karmakar, and Dhiman Saha. PRINCE under differential fault attack: Now in 3D. In Chip-Hong Chang, Ulrich Rührmair, Stefan Katzenbeisser, and Patrick Schaumont, editors, *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2020*, pages 81–91. ACM, 2020.

[BBB+21] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Mustafa Khairallah, Thomas Peyrin, Sumanta Sarkar, and Siang Meng Sim. DEFAULT: cipher level resistance against differential fault attack. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security*, volume 13091 of *Lecture Notes in Computer Science*, pages 124–156. Springer, 2021.

[BBC+23] Anubhab Baksi, Jakub Breier, Anupam Chattopadhyay, Tomas Gerlich, Sylvain Guilley, Naina Gupta, Kai Hu, Takanori Isobe, Arpan Jati, Petr Jedlicka, Hyunjun Kim, Fukang Liu, Zdenek Martinasek, Kosei Sakamoto, Hwajeong Seo, Rentaro Shiba, and Ritu Ranjan Shrivastwa. BAKSHEESH: similar yet different from GIFT. *IACR Cryptol. ePrint Arch.*, page 750, 2023.

[BCI+20] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB. *IACR Cryptol. ePrint Arch.*, page 738, 2020.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptol.*, 14(2):101–119, 2001.

[BK06] Johannes Blömer and Volker Krummel. Fault based collision attacks on AES. In Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert, editors, *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006*, volume 4236 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2006.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BPP+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 321–345. Springer, 2017.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997.

[DEG+18] Christoph Dobraunig, Maria Eichlseder, Hannes Groß, Stefan Mangard, Florian Mendel, and Robert Primas. Statistical ineffective fault attacks on masked AES with fault countermeasures. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security*, volume 11273 of *Lecture Notes in Computer Science*, pages 315–342. Springer, 2018.

[DFL11] Patrick Derbez, Pierre-Alain Fouque, and Delphine Leresteux. Meet-in-the-middle and impossible differential fault analysis on AES. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop*, volume 6917 of *Lecture Notes in Computer Science*, pages 274–291. Springer, 2011.

[DP20] Patrick Derbez and Léo Perrin. Meet-in-the-middle attacks and structural analysis of round-reduced PRINCE. *J. Cryptol.*, 33(3):1184–1215, 2020.

[FJLT13] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In Wieland Fischer and Jörn-Marc Schmidt, editors, *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 108–118. IEEE Computer Society, 2013.

[GSSC15] Shamit Ghosh, Dhiman Saha, Abhrajit Sengupta, and Dipanwita Roy Chowdhury. Preventing fault attacks using fault randomization with a case study on AES. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy - 20th Australasian Conference, ACISP 2015*, volume 9144 of *Lecture Notes in Computer Science*, pages 343–355. Springer, 2015.

[JKP24] Amit Jana, Anup Kumar Kundu, and Goutam Paul. More vulnerabilities of linear structure sbox-based ciphers reveal their inability to resist DFA. In Kai-Min Chung and Yu Sasaki, editors, *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, Proceedings, Part VIII*, volume 15491 of *Lecture Notes in Computer Science*, pages 168–203. Springer, 2024.

[KAKS22] Anup Kumar Kundu, Aikata, Banashri Karmakar, and Dhiman Saha. Fault analysis of the PRINCE family of lightweight ciphers. *J. Cryptogr. Eng.*, 12(4):475–494, 2022.

[LCMW21] Haoxiang Luo, Weijian Chen, Xinyue Ming, and Yifan Wu. General differential fault attack on PRESENT and GIFT cipher with nibble. *IEEE Access*, 9:37697–37706, 2021.

[LGH22] Shuai Liu, Jie Guan, and Bin Hu. Fault attacks on authenticated encryption modes for GIFT. *IET Inf. Secur.*, 16(1):51–63, 2022.

[MFJ21] XIE Min, TIAN Feng, and LI Jiaqi. Differential fault attack on GIFT. *Chinese Journal of Electronics*, 30(4):669–675, 2021.

[NDE22] Marcel Nageler, Christoph Dobraunig, and Maria Eichlseder. Information-combining differential fault attacks on DEFAULT. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings, Part III*, volume 13277 of *Lecture Notes in Computer Science*, pages 168–191. Springer, 2022.

[SC16] Dhiman Saha and Dipanwita Roy Chowdhury. Encounter: On breaking the nonce barrier in differential fault analysis with a case-study on PAEQ. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 581–601. Springer, 2016.

[SSL15] Kazuo Sakiyama, Yu Sasaki, and Yang Li. *Security of Block Ciphers - From Algorithm Design to Hardware Implementation*. Wiley, 2015.

[Tec17] N.I.N.I.S. Technology. *Report on Lightweight Cryptography: NiSTIR 8114*. CreateSpace Independent Publishing Platform, 2017.

[VZB+22] Navid Vafaei, Sara Zarei, Nasour Bagheri, Maria Eichlseder, Robert Primas, and Hadi Soleimany. Statistical effective fault attacks: The other side of the coin. *IEEE Trans. Inf. Forensics Secur.*, 17:1855–1867, 2022.

[ZLZ+18] Fan Zhang, Xiaoxuan Lou, Xinjie Zhao, Shivam Bhasin, Wei He, Ruyi Ding, Samiya Qureshi, and Kui Ren. Persistent fault analysis on block ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):150–172, 2018.

A Predecessor Sets for Truncated Differentials of GIFT-128 and BAKSHEESH

$$\mathcal{P}_{128}^{\Delta_1} = \left\{ \begin{array}{l} \overbrace{\begin{array}{l}[0000\ 000b\ 0000\ 000d\ 0000\ 000e\ 0000\ 0007] \\ [0000\ 0007\ 0000\ 000b\ 0000\ 000d\ 0000\ 000e] \\ [0000\ 00b0\ 0000\ 00d0\ 0000\ 00e0\ 0000\ 0070] \\ [0000\ 0070\ 0000\ 00b0\ 0000\ 00d0\ 0000\ 00e0] \\ [0000\ 0b00\ 0000\ 0d00\ 0000\ 0e00\ 0000\ 0700] \\ [0000\ 0700\ 0000\ 0b00\ 0000\ 0d00\ 0000\ 0e00] \\ [0000\ b000\ 0000\ d000\ 0000\ e000\ 0000\ 7000] \\ [0000\ 7000\ 0000\ b000\ 0000\ d000\ 0000\ e000] \\ [000b\ 0000\ 000d\ 0000\ 000e\ 0000\ 0007\ 0000] \\ [0007\ 0000\ 000b\ 0000\ 000d\ 0000\ 000e\ 0000] \\ [00b0\ 0000\ 00d0\ 0000\ 00e0\ 0000\ 0070\ 0000] \\ [0070\ 0000\ 00b0\ 0000\ 00d0\ 0000\ 00e0\ 0000] \\ [0b00\ 0000\ 0d00\ 0000\ 0e00\ 0000\ 0700\ 0000] \\ [0700\ 0000\ 0b00\ 0000\ 0d00\ 0000\ 0e00\ 0000] \\ [b000\ 0000\ d000\ 0000\ e000\ 0000\ 7000\ 0000] \\ [7000\ 0000\ b000\ 0000\ d000\ 0000\ e000\ 0000] \end{array}}^{\text{Type-I}} \quad \overbrace{\begin{array}{l}[0000\ 0086\ 0000\ 0043\ 0000\ 0029\ 0000\ 001c] \\ [0000\ 0094\ 0000\ 00c2\ 0000\ 0061\ 0000\ 0038] \\ [0000\ 0860\ 0000\ 0430\ 0000\ 0290\ 0000\ 01c0] \\ [0000\ 0940\ 0000\ 0c20\ 0000\ 0610\ 0000\ 0380] \\ [0000\ 8600\ 0000\ 4300\ 0000\ 2900\ 0000\ 1c00] \\ [0000\ 9400\ 0000\ c200\ 0000\ 6100\ 0000\ 3800] \\ [0086\ 0000\ 0043\ 0000\ 0029\ 0000\ 001c\ 0000] \\ [0094\ 0000\ 00c2\ 0000\ 0061\ 0000\ 0038\ 0000] \\ [0860\ 0000\ 0430\ 0000\ 0290\ 0000\ 01c0\ 0000] \\ [0940\ 0000\ 0c20\ 0000\ 0610\ 0000\ 0380\ 0000] \\ [8600\ 0000\ 4300\ 0000\ 2900\ 0000\ 1c00\ 0000] \\ [9400\ 0000\ c200\ 0000\ 6100\ 0000\ 3800\ 0000] \end{array}}^{\text{Type-II}} \end{array} \right\}$$

$$\mathcal{P}_{128}^{\Delta_2} = \left\{ \begin{array}{l} \overbrace{\begin{array}{l}[0808\ 0808\ 0404\ 0404\ 0202\ 0202\ 0101\ 0101] \\ [0101\ 0101\ 0808\ 0808\ 0404\ 0404\ 0202\ 0202] \\ [0202\ 0202\ 0101\ 0101\ 0808\ 0808\ 0404\ 0404] \\ [0404\ 0404\ 0202\ 0202\ 0101\ 0101\ 0808\ 0808] \\ [8080\ 8080\ 4040\ 4040\ 2020\ 2020\ 1010\ 1010] \\ [1010\ 1010\ 8080\ 8080\ 4040\ 4040\ 2020\ 2020] \\ [2020\ 2020\ 1010\ 1010\ 8080\ 8080\ 4040\ 4040] \\ [4040\ 4040\ 2020\ 2020\ 1010\ 1010\ 8080\ 8080] \end{array}}^{\text{Type-I}} \quad \overbrace{\begin{array}{l}[0909\ 0909\ 0c0c\ 0c0c\ 0606\ 0606\ 0303\ 0303] \\ [0303\ 0303\ 0909\ 0909\ 0c0c\ 0c0c\ 0606\ 0606] \\ [0606\ 0606\ 0303\ 0303\ 0909\ 0909\ 0c0c\ 0c0c] \\ [9090\ 9090\ c0c0\ c0c0\ 6060\ 6060\ 3030\ 3030] \\ [3030\ 3030\ 9090\ 9090\ c0c0\ c0c0\ 6060\ 6060] \\ [6060\ 6060\ 3030\ 3030\ 9090\ 9090\ c0c0\ c0c0] \end{array}}^{\text{Type-II}} \end{array} \right\}$$

B Guide to Reproduce the ChipWhisperer Experiments

This guide is with reference to the following repository[5]. The following two commands are sufficient to build the necessary dependencies, induce faults, and perform final key recovery.

```
$ python setup.py build_ext --inplace
$ python3 attack.py
```

ToolChain: To mount the attack on the device, we use the ChipWhisperer open-source toolchain. Please ensure this toolchain is set up before proceeding with the attack. The folder `Setup_Scripts` and the `TOFA` folder pertain to this toolchain's requirements. The contents of the `TOFA` folder build the necessary `.hex` files required for programming the microcontroller. To do this, run a `make` command in the `TOFA` folder.

Fault Injection: The script used to induce the fault is `TOFA_FA.ipynb`. This Jupyter Notebook runs the GIFT-128 encryption and injects faults via clock glitching. It can be opened and run to see how this is done. Note that the script will only run in the presence of the hardware (CW1173).

Key-Recovery: The Jupyter Notebook- `TOFA_FA.ipynb` is called via Python script- `attack.py`. This Python file collects the faulty and correct ciphertexts and returns the number of keys recovered. In the parameters set, a unique key is obtained. The parameters within this file responsible for the number of faults `num_faults` can be modified to see how the key recovery works for fewer faults.

[5] https://github.com/ShibamCrS/Tofa/tree/main/gift128