IISFA: Ineffective Instruction-Skip Fault Attacks on Bitslice Implementations

Hanbeom Shin[1], Dongwoo Kang[1], Donggeun Kwon[2], Seokhie Hong[3] and HeeSeok Kim[1]

[1] Korea University, Republic of Korea
newonetiger@korea.ac.kr, redr1102@korea.ac.kr, 80khs@korea.ac.kr
[2] Kunsan National University, Republic of Korea
dgkwon@kunsan.ac.kr
[3] SmartM2M, Republic of Korea
shhong@smartm2m.co.kr

Abstract. Redundancy countermeasures prevent differential fault analysis by withholding faulty ciphertexts. Ineffective fault analysis and statistical ineffective fault attacks exploit executions in which an injected fault leaves the result unchanged, using constraints on intermediate values or statistical bias to recover keys. In this paper, we present an Ineffective Instruction-Skip Fault Attack (IISFA) on bitsliced implementations protected by redundancy. A bitsliced instruction processes m S-boxes simultaneously. Skipping it leaves the group unchanged exactly when all m S-box inputs belong to its ineffective set, which we compute from the implementation. Each ciphertext output by the device therefore restricts all m n-bit subkeys to candidates whose S-box inputs belong to that set. The ineffective probability specifies how often a uniformly random S-box input makes the skip ineffective, and its m-th power gives the ciphertext output probability for independent inputs. We determine which key candidates remain indistinguishable under a single target instruction. We give key-recovery procedures for both known and chosen plaintexts and combine different target instructions across queries, with one instruction skipped per execution. We calculate expected injection counts or enclosing intervals for the known- and chosen-plaintext procedures from the ineffective sets and use them to reduce the search for efficient target sequences. We evaluate five bitsliced implementations of `GIFT`, `SKINNY` and `PIPO` on an 8-bit AVR microcontroller under a redundancy countermeasure using clock glitches. All measured key-recovery campaigns succeed in every trial.

Keywords: Fault attack · Instruction skip · Ineffective fault · Bitslice implementation · Redundancy countermeasure · PIPO · GIFT · SKINNY

1 Introduction

A fault attack disturbs a cryptographic device while it is running and derives information about its secrets from how the disturbance changes its behaviour. The disturbance can be produced by a laser pulse [SA03], by lowering the supply voltage [BBB+10, BFP19], or by glitching the clock [ESH+11], and the equipment needed is within reach of an attacker who holds the device. Its effect is usually abstracted into a fault model, in which a bit is flipped, a value is stuck at a constant, a word is randomised, or an instruction is not executed. Among these, we work with the last. On a microcontroller running a software implementation, a clock glitch narrow enough to violate the setup time of a single pipeline stage makes the processor skip one instruction while leaving the rest of the execution intact [MHER14], and the instruction that is skipped is determined by when the glitch

arrives. Consequently, the instruction-skip model is both realistic and precise, as the attacker chooses a point in the instruction stream rather than a value in a register.

Differential Fault Analysis (DFA), introduced by Biham and Shamir [BS97], uses differences between correct and faulty outputs to restrict secret-key candidates. At the cipher-design level, `DEFAULT` was designed to resist DFA through properties of its S-boxes [BBB+21]. However, at EUROCRYPT 2022, Nageler et al. [NDE22] challenged this resistance with information-combining attacks that combine key constraints across rounds. Subsequently, at ASIACRYPT 2024, Jana et al. [JKP24] improved DFA using differential trails with a single solution. More recently, ToFA [KGAS25] uses truncated impossible differentials to eliminate incorrect key candidates for `GIFT` and `BAKSHEESH`. MIFA [SKK+26] uses Mixed-Integer Linear Programming (MILP) to identify differential trails with a single solution for key recovery in deeper rounds.

Other fault attacks exploit different properties of faulty outputs. Specifically, statistical fault analysis [FJLT13] recovers keys from biased faulty values without requiring correct ciphertexts, while persistent fault analysis [ZLZ+18] corrupts a stored S-box entry and analyses the resulting ciphertext distribution. Division-property fault attacks [KGSR23] use fault-induced invariants to reduce the key space of `PRESENT` and `GIFT`. Although these approaches differ in their analysis and fault models, they all require faulty ciphertexts.

Redundancy countermeasures protect against transient-fault versions of these attacks by detecting faults before the device outputs a ciphertext. A simple scheme encrypts the same input twice under the same key and compares the resulting ciphertexts [DEK+18]. The device outputs a ciphertext only when the two results match. If a fault changes only one computation, the comparison fails and no ciphertext is returned. This prevents the analysis of faulty ciphertexts, provided that the comparison and output decision are not faulted. However, the distinction between a ciphertext output by the device and no output remains observable. Consequently, withholding faulty ciphertexts does not remove all information about an injected fault.

Ineffective Fault Analysis (IFA), introduced by Clavier [Cla07], exploits faulted executions whose outputs remain correct. His attack on DES assumes a stuck-at-zero fault model that forces the 8-bit result of a targeted XOR instruction to zero, regardless of its operands. Under this model, an unchanged ciphertext reveals a constraint on the intermediate value. Statistical Ineffective Fault Attacks (SIFA) [DEK+18] instead exploit the biased distribution of intermediate values in executions with ineffective faults, without requiring an exact model of the fault effect. Subsequent work demonstrated SIFA on implementations of the Advanced Encryption Standard (AES) protected by both masking and redundancy [DEG+18]. We focus on redundancy-protected implementations under the instruction-skip fault model, with the constraints computed directly from the bitsliced implementation.

The implementations we attack are bitsliced. The state is stored so that each register holds one bit position of many S-boxes at once, and the S-box is computed by a branch-free sequence of bitwise instructions [Bih97]. Bitslicing and refinements such as fixslicing support efficient constant-time software implementations [ANP20, AP21]. A single instruction therefore processes m S-boxes in parallel, m being the register width. Skipping one of these instructions leaves a set of S-box inputs whose output is unchanged, and the device outputs a ciphertext exactly when all m inputs handled by that instruction fall in this set. Consequently, whether the device outputs a ciphertext gives information about the inputs processed by the skipped instruction.

Our Contributions

In this paper, we present an Ineffective Instruction-Skip Fault Attack (IISFA) on bitsliced implementations protected by a redundancy countermeasure. The SDDT framework [KSK+26] recovers keys from faulty ciphertexts by analysing skip-induced output

differences. IISFA instead uses only whether the device outputs a ciphertext after the skip. Our contributions are summarised as follows.

- **Key recovery from ineffective instruction skips.** We show that keys can be recovered from bitsliced implementations using only whether the device outputs a ciphertext after an instruction skip. For this purpose, we define the ineffective set as the S-box inputs whose outputs remain unchanged when the target instruction is skipped. Each ciphertext output by the device restricts the key candidates to those whose S-box inputs belong to this set. The ineffective probability determines the ciphertext output probability under uniform, independent S-box inputs. Furthermore, we determine the key candidates that a single target cannot distinguish through its set-preserving XOR differences. These properties identify instructions that permit single-target key recovery under the redundancy countermeasure.

- **Key recovery with known and chosen plaintexts.** Based on this, we give key-recovery procedures for both known and chosen plaintexts. Furthermore, we combine the key constraints obtained by targeting different instructions across separate encryption executions. This combination can distinguish key candidates that remain indistinguishable under a single target. Each execution still skips exactly one instruction, and constraints from three or more target instructions can be combined in the same way. We use each target until it cannot distinguish any remaining candidates before switching to the next target. Single-target attacks and instruction combinations are evaluated separately in both plaintext settings.

- **Efficient target sequence selection.** We propose a method for selecting efficient target instruction sequences for known- and chosen-plaintext attacks. Specifically, we combine dynamic programming with pruning based on intervals for the expected injection counts calculated from the ineffective sets. Across the five implementations and both plaintext settings, this reduces the search space by 98.4–99.98 %, without limiting sequences to two or three targets.

- **Experimental evaluation.** We evaluate the attack on an 8-bit AVR microcontroller running five bitsliced implementations of `GIFT`, `SKINNY` and `PIPO` under a redundancy countermeasure, with faults injected by clock glitching. Table 1 summarises the single-target and multiple-target attacks, which recovered the round keys in all ten trials for every measured condition. Furthermore, combining target instructions reduced the mean injection count by 16–36 % with known plaintexts and 14–22 % with chosen plaintexts on both `GIFT` implementations and `SKINNY`-64/64. The single-instruction attack and the instruction sequences are evaluated in separate campaigns, and every measured injection count agrees with the same-seed simulation trial for trial.

The artifact is available at `https://anonymous.4open.science/r/IISFA/`.

2 Preliminaries

2.1 Instruction Skip Faults and Redundancy Countermeasures

We assume an attacker who can make the device skip one instruction of its choice in an execution, and who learns nothing about the execution except whether it produced a ciphertext. Concretely, the attacker triggers a disturbance at a chosen cycle of the instruction stream. The instruction issued in that cycle does not take effect, and every other instruction executes normally. We realise this in Section 5 by glitching the clock,

Table 1: Mean injection counts and successful round-key recoveries measured on the device under a redundancy countermeasure. Single and Multiple denote one target instruction and a sequence of target instructions, respectively. Each measured condition has ten trials, and a successful recovery determines all targeted round keys.

	Known plaintext		Chosen plaintext		
Implementation	Single	Multiple	Single	Multiple	Recoveries
`GIFT`-64/128	1858	1308	586	506	10/10
`GIFT`-128/128	2022	1289	610	511	10/10
`SKINNY`-64/64	962	809	319	248	10/10
`SKINNY`-128/128	1380	–	430	–	10/10
`PIPO`-64/128	550	–	213	–	10/10

–: The tested multiple-target sequences required more injections on average than the single target in simulation, so only single-target device results are reported. Recoveries apply to every measured condition in the row.

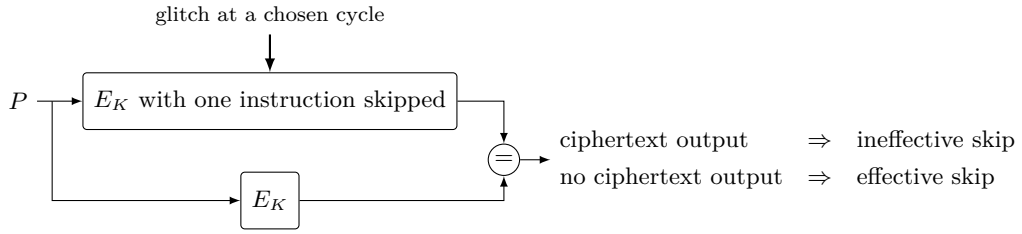

Figure 1: Encryption implementation targeted by IISFA under a redundancy countermeasure. One instruction is skipped in one of the two encryptions, and the device outputs a ciphertext only when their results agree.

which is the cheapest of the available means and the one for which single instruction skips on microcontrollers are best documented [MHER14].

The countermeasure we attack is redundancy in its simplest form. The device computes the same encryption twice, compares the two results, and transmits the ciphertext only if they are equal, and otherwise transmits nothing. This is what makes an instruction skip hard to use in the usual way, because whenever the skip changes the result of the first computation, the comparison fails and the faulty ciphertext never leaves the device. We assume the comparison itself is not a target, so the attacker spends its one skip inside the cipher.

The attacker therefore holds an oracle that answers a single bit, drawn in Figure 1. It submits a plaintext and a cycle at which to inject, and observes either a ciphertext or no ciphertext output. No ciphertext output means the skip changed the encryption result. A ciphertext means it did not, and that ciphertext is the correct one, carrying no fault information of its own.

We consider two settings for the plaintexts. A *known-plaintext* attacker observes the plaintexts but cannot influence them, and a *chosen-plaintext* attacker selects them. In both settings the key is fixed across queries and the attacker may query as often as it likes.

2.2 Bitslice Implementations

Let the S-box be a permutation S of $\{0,1\}^n$. A bitsliced implementation stores the state transposed, so that register j holds bit j of many S-box inputs at once, one per bit position of the register. We call the content of register j a *slice* and each bit position a *lane*, and

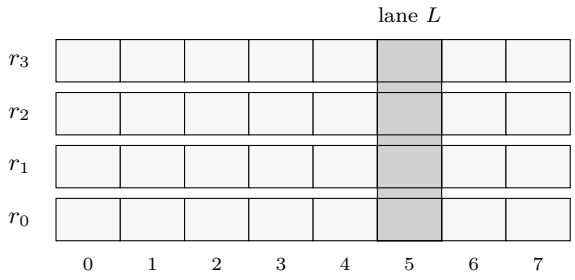

Figure 2: Bitslice representation of eight 4-bit S-box inputs. Register r_j holds bit j of all eight inputs, and each column contains one S-box input.

write m for the number of lanes, which is the register width of the platform. A group of n registers thus holds the inputs of m S-boxes, and a state of N bits is covered by $N/(nm)$ such groups. Figure 2 illustrates this representation for eight 4-bit S-box inputs.

The S-box is then computed by a fixed sequence of bitwise instructions on those n registers and a few temporaries, with no branches and no table lookups. We write $\mathrm{op}_1, \ldots, \mathrm{op}_L$ for these instructions in execution order. Because every bitwise instruction applies the same operation to all m lanes at once, one execution computes m S-boxes in parallel, and one skipped instruction is skipped in all m of them. This is the property the whole paper rests on.

It also means the lanes do not interact. During the S-box computation, the value a lane holds at any point is a function of that lane's S-box input alone, so whether a skipped instruction changes the output of a given lane depends on that lane alone. Skipping an instruction therefore leaves the S-box outputs unchanged if and only if it leaves them unchanged in every lane separately. Any diffusion between lanes happens in the linear layer, after the S-box computation has finished.

2.3 Target Ciphers

The target variants are `GIFT`-64, `GIFT`-128, `SKINNY`-64, `SKINNY`-128, and `PIPO`-64. Our round-key recovery method applies to all supported key lengths of these variants, since their S-box functions are independent of the key length. The key schedule determines how many round keys must be recovered to obtain the master key. For a fixed block size, the `SKINNY` and `PIPO` variants retain the same S-box computation across key lengths, while their round counts and key schedules change. `GIFT` [BPP+17] uses a bit-permutation linear layer, and `SKINNY` [BJK+16] is a tweakable block cipher. Both have S-boxes that can be computed by bitwise operations. `PIPO` [KJK+20] was designed with bitsliced software in mind, and its S-box is specified directly as a sequence of such operations. The five experimental implementations cover 4-bit and 8-bit S-boxes and analysed instruction sequences of lengths 23 to 74.

Listing 1 gives the AVR instructions used for the `GIFT` and `SKINNY` S-boxes in our experiments. The `GIFT` sequence is shared by both block sizes. To make the slice operations explicit, we use register aliases `s0`, `s1`, and so on for the registers loaded with input slices 0, 1, and so on, and `t` for the temporary register. These names remain attached to the same registers as their values are updated. Each panel gives the corresponding AVR registers. For example, instructions 5–7 in panel (a) compute $s_1 \leftarrow s_1 \oplus (s_0 \wedge s_2)$ through `t`. The final stores specify the output slice order. The numbers are the instruction indices used throughout the analysis. Listing 2 gives the `PIPO` implementation in the same notation. It uses eight input slice registers and four temporary registers, named `t0`, `t1`, `t2`, and `t`.

`GIFT` and `PIPO` use bit permutations as their linear layers, whereas `SKINNY` uses ShiftRows followed by MixColumns. Write Y_r for the state entering round r, arranged as

Listing 1: AVR bitslice S-box implementations with symbolic register aliases. Panels (a), (b), and (c) give the `GIFT`, `SKINNY-64`, and `SKINNY-128` S-boxes, respectively. The names `s0`, `s1`, and so on identify the input slice registers, and `t` is temporary. Instruction indices count only the instructions shown. Z points to the slice array.

(a) `GIFT`

```
// s0=r18, s1=r19
// s2=r20, s3=r21
// t=r22 (temporary)

 1 ld   s0, Z
 2 ldd  s1, Z+1
 3 ldd  s2, Z+2
 4 ldd  s3, Z+3
 5 mov  t, s0
 6 and  t, s2
 7 eor  s1, t
 8 mov  t, s1
 9 and  t, s3
10 eor  s0, t
11 mov  t, s0
12 or   t, s1
13 eor  s2, t
14 eor  s3, s2
15 eor  s1, s3
16 com  s3
17 mov  t, s0
18 and  t, s1
19 eor  s2, t
20 st   Z, s3
21 std  Z+1, s1
22 std  Z+2, s2
23 std  Z+3, s0
```

(b) `SKINNY-64`

```
// s0=r18, s1=r19
// s2=r20, s3=r21
// t=r22 (temporary)

 1 ld   s0, Z
 2 ldd  s1, Z+1
 3 ldd  s2, Z+2
 4 ldd  s3, Z+3
 5 mov  t, s2
 6 or   t, s3
 7 com  t
 8 eor  s0, t
 9 mov  t, s1
10 or   t, s2
11 com  t
12 eor  s3, t
13 mov  t, s0
14 or   t, s1
15 com  t
16 eor  s2, t
17 mov  t, s3
18 or   t, s0
19 com  t
20 eor  s1, t
21 st   Z, s1
22 std  Z+1, s2
23 std  Z+2, s3
24 std  Z+3, s0
```

(c) `SKINNY-128`

```
// s0=r18, s1=r19, s2=r20, s3=r21
// s4=r22, s5=r23, s6=r24, s7=r25
// t=r26 (temporary)

 1 ld   s0, Z          25 mov  t, s0
 2 ldd  s1, Z+1        26 or   t, s3
 3 ldd  s2, Z+2        27 com  t
 4 ldd  s3, Z+3        28 eor  s1, t
 5 ldd  s4, Z+4        29 mov  t, s6
 6 ldd  s5, Z+5        30 or   t, s5
 7 ldd  s6, Z+6        31 com  t
 8 ldd  s7, Z+7        32 eor  s7, t
 9 mov  t, s7          33 mov  t, s5
10 or   t, s6          34 or   t, s4
11 com  t              35 com  t
12 eor  s4, t          36 eor  s3, t
13 mov  t, s3          37 mov  t, s1
14 or   t, s2          38 or   t, s7
15 com  t              39 com  t
16 eor  s0, t          40 eor  s2, t
17 mov  t, s2          41 st   Z, s2
18 or   t, s1          42 std  Z+1, s7
19 com  t              43 std  Z+2, s6
20 eor  s6, t          44 std  Z+3, s1
21 mov  t, s4          45 std  Z+4, s3
22 or   t, s0          46 std  Z+5, s0
23 com  t              47 std  Z+6, s4
24 eor  s5, t          48 std  Z+7, s5
```

`ld` loads a byte from address Z into the first operand register, while `ldd` loads from $\mathtt{Z}+q$ for a byte offset q. `st` stores the second operand register at Z, and `std` stores it at $\mathtt{Z}+q$. `mov` copies the second operand into the first. `and`, `or`, and `eor` combine the two operand values by bitwise AND, OR, and XOR, respectively, and write the result to the first operand. `com` complements every bit of its operand.

cells of n bits, one cell per S-box. Then

$$Y_{r+1} \;=\; A_r(Y_r) \oplus K_r, \tag{1}$$

where A_r is the invertible transformation before the XOR with K_r and K_r is the key contribution at the round output. `GIFT` and `PIPO` XOR the round key directly at this point. For `SKINNY`, K_r is the equivalent key obtained by applying ShiftRows and MixColumns to the round subtweakey, since the subtweakey is added before these linear operations. Public round constants are included in A_r or K_r. Recovering K_r for enough rounds gives the master key through the key schedule. The implementations fix $m = 8$ lanes, so one group of registers holds eight S-boxes and a round is covered by G groups.

Round indices start at $r = 1$. The input whitening of `PIPO` has index $r = 0$, with Y_0 the plaintext, A_0 the identity, and K_0 the whitening key. Within each S-box implementation, instruction indices $1 \le i \le L$ follow execution order.

2.4 Skip Difference Distribution Table

Kang et al. [KSK+26] study instruction skips in bitsliced S-box implementations through what they call the skip difference distribution table. Writing $S^{(i)}$ for the function the

Listing 2: AVR bitslice S-box implementation of `PIPO` with symbolic register aliases. The instruction sequence continues from the left column to the right. Instruction indices count only the instructions shown. Z points to the slice array.

```
                                        PIPO
// s0=r18, s1=r19, s2=r20, s3=r21
// s4=r22, s5=r23, s6=r24, s7=r25
// t0=r26, t1=r27, t2=r16, t=r17 (temporaries)

 1 ld   s0, Z              26 eor  s2, t             51 mov  t, t1
 2 ldd  s1, Z+1            27 mov  t, s2             52 and  t, t0
 3 ldd  s2, Z+2            28 or   t, s1             53 eor  t2, t
 4 ldd  s3, Z+3            29 eor  s0, t             54 eor  s2, t0
 5 ldd  s4, Z+4            30 mov  t, s2             55 mov  t0, s1
 6 ldd  s5, Z+5            31 or   t, s0             56 eor  t0, t2
 7 ldd  s6, Z+6            32 eor  s1, t             57 mov  s1, s0
 8 ldd  s7, Z+7            33 com  s2                58 eor  s1, t1
 9 mov  t, s7              34 eor  s7, s1            59 mov  s0, s7
10 and  t, s6              35 eor  s3, s2            60 mov  s7, t0
11 eor  s5, t              36 eor  s4, s0            61 mov  t, s3
12 mov  t, s3              37 mov  t0, s7            62 mov  s3, s6
13 and  t, s5              38 mov  t1, s3            63 mov  s6, t
14 eor  s4, t              39 mov  t2, s4            64 mov  t, s4
15 eor  s7, s4             40 mov  t, t0             65 mov  s4, s5
16 eor  s6, s3             41 and  t, s5             66 mov  s5, t
17 mov  t, s4              42 eor  s6, t             67 st   Z, s0
18 or   t, s5              43 eor  t0, s6            68 std  Z+1, s1
19 eor  s3, t              44 mov  t, t2             69 std  Z+2, s2
20 eor  s5, s7             45 or   t, t1             70 std  Z+3, s3
21 mov  t, s5              46 eor  s6, t             71 std  Z+4, s4
22 and  t, s6              47 eor  t1, s5            72 std  Z+5, s5
23 eor  s4, t              48 mov  t, s6             73 std  Z+6, s6
24 mov  t, s1              49 or   t, t2             74 std  Z+7, s7
25 and  t, s0              50 eor  s5, t
```

implementation computes when its i-th instruction is skipped, the table records, for each instruction index i and each output difference Δ, how many inputs x satisfy $S^{(i)}(x) \oplus S(x) = \Delta$. Their attack reads the key off the rows with $\Delta \neq 0$. A faulty ciphertext reveals the difference the skip produced, the table says which inputs can produce it, and the attacker eliminates inconsistent key candidates to recover the key.

However, under a redundancy countermeasure those rows are unreachable. An execution whose output differs from the unfaulted one is exactly an execution the comparison rejects, so no ciphertext is transmitted and the attacker receives no ciphertext. What remains observable is the row the previous attack has no use for, $\Delta = 0$.

3 Ineffective Sets and Target Instruction Selection

3.1 IISFA

IISFA exploits the fact that the effect of an instruction skip depends on the data being processed. For example, consider `and rA, rB` and let a and b be the bits in the two registers for one lane immediately before the instruction. Normal execution replaces a with $a \wedge b$, whereas skipping leaves a unchanged. If $a = 0$ and $b = 1$, the AND result is $0 \wedge 1 = 0$, and skipping also leaves 0, so the results agree. If $a = 1$ and $b = 0$, the AND result is $1 \wedge 0 = 0$, but skipping leaves 1, so the results differ. The other two pairs, $(0, 0)$ and $(1, 1)$, also leave the destination bit unchanged.

Table 2: Parameters of the five implementations. N is the block size, n the S-box size, G the number of groups of eight S-boxes per round, and L the number of analysed instructions. Slices counts the 8-bit input slices per group, excluding temporary registers. The register count includes input and temporary data registers and excludes the pointer Z (r30:r31). S-boxes counts S-boxes per round.

Implementation	N	Key	n	Rounds	S-boxes	G	Slices	Registers	L
GIFT-64/128	64	128	4	28	16	2	4	5	23
GIFT-128/128	128	128	4	40	32	4	4	5	23
SKINNY-64/64	64	64	4	32	16	2	4	5	24
SKINNY-128/128	128	128	8	40	16	2	8	9	48
PIPO-64/128	64	128	8	13	8	1	8	12	74

In a bitsliced implementation, one instruction operates on m S-boxes simultaneously. After a skip, ciphertext output means that every affected S-box has the same final output as in normal execution. IISFA uses this condition to remove incorrect subkey candidates. At the first S-box layer under attack, each input is formed by XORing a value computed from the plaintext with an unknown subkey. The attacker substitutes a subkey candidate and computes the corresponding S-box input. Using this candidate input, the attacker simulates the S-box both normally and with the target instruction skipped. If the device outputs a ciphertext but the two simulated outputs differ, the subkey candidate is discarded. Under that candidate, the skip would make the two redundant encryptions disagree, so the device would output no ciphertext. The same test is applied to each of the m S-boxes. When the target instruction satisfies the single-target recovery conditions in Section 3.4, repeating this test for different plaintexts can recover each subkey.

3.2 Ineffective Set of a Skipped Instruction

Let σ denote the register values left by the preceding code, apart from the S-box input. Write $S_\sigma^{(i)}$ for the function computed when instruction i is skipped and the remaining instructions execute normally with those register values.

Definition 1 (Ineffective set)**.** The *ineffective set* of instruction i with register values σ left by the preceding code is $\mathcal{Z}_i(\sigma) = \{\, x \in \{0,1\}^n : S_\sigma^{(i)}(x) = S(x) \,\}$. Its *ineffective probability* is $p_i(\sigma) = |\mathcal{Z}_i(\sigma)|/2^n$, the probability that skipping instruction i is ineffective for one uniformly random S-box input.

Skipping a write can expose the previous value of its destination register to a later instruction. The observation model in Section 2.1 does not give the attacker access to these register values. Moreover, a candidate input of the targeted S-box does not in general determine σ, since preceding code may leave values from computations involving other unknown subkeys. For a σ-dependent target, checking a candidate for one subkey may therefore require candidates for those other subkeys as well. We call a target *σ-free* when its ineffective set is independent of the register values left by the preceding code. We use these targets so that each n-bit subkey candidate can be checked without determining those unknown values. Its ineffectiveness condition then depends only on the S-box input, and the resulting constraint remains valid for every possible value of σ. All targets evaluated here are σ-free, so we write $\mathcal{Z}_i$ and p_i without the argument.

Suppose the attacker skips instruction i while a group of m S-boxes is being computed. Because the implementation treats the lanes independently (Section 2.2), the output of the group is unchanged if and only if each of the m S-box inputs lies in $\mathcal{Z}_i$. Everything after the group is then computed from the correct state. Thus the two redundant computations

agree and the device outputs the ciphertext. Consequently, if the inputs of the group are uniform and independent, the ciphertext output probability is

$$\Pr[\text{ciphertext output}] \;=\; p_i^m.$$

Under these sampling assumptions, the expected number of injections per output ciphertext is p_i^{-m}, which increases with m for $0 < p_i < 1$. With $m = 8$ lanes, $p_i = 3/4$ gives one output event in about ten injections while $p_i = 1/2$ gives one in 256.

An output event is informative because the attacker can compute the state immediately before subkey addition from the plaintext and previously recovered round keys. Let v be the cell value of $A_{r-1}(Y_{r-1})$ in the targeted lane, before adding the n-bit subkey k from K_{r-1}. By (1), the actual S-box input is $x = v \oplus k$. Thus v is known, while x is unknown until k is recovered. Ciphertext output therefore implies $k \in v \oplus \mathcal{Z}_i$ for every lane. Consequently, one output event confines each of the m n-bit subkeys of the group to a set of $|\mathcal{Z}_i|$ values, and further output events intersect these sets. The absence of ciphertext also provides information by excluding key combinations in which all m lanes would have been ineffective. It does not imply $k \notin v \oplus \mathcal{Z}_i$ for every lane, since one effective lane suffices to prevent output. Retaining this information generally requires constraints on combinations of subkey values. The known-plaintext procedure in Section 4.1 uses output events to keep its updates as independent intersections for each n-bit subkey. The chosen-plaintext procedure in Section 4.2 isolates one lane and uses both outcomes.

3.3 Set-Preserving XOR Differences

Combining the key-candidate constraints obtained from repeated queries does not always uniquely determine an n-bit subkey, no matter how many queries are made. The obstruction is a symmetry of the ineffective set of Definition 1.

Definition 2 (Set-preserving XOR differences)**.** The *set-preserving XOR differences* of $\mathcal{Z}_i$ form the set $\mathcal{D}(\mathcal{Z}_i) = \{\, d \in \{0,1\}^n : \mathcal{Z}_i \oplus d = \mathcal{Z}_i \,\}$. Here d is an n-bit difference, and $\mathcal{Z}_i \oplus d$ denotes $\{x \oplus d : x \in \mathcal{Z}_i\}$.

For a difference $d \in \mathcal{D}(\mathcal{Z}_i)$, XORing any input with d leaves its membership in $\mathcal{Z}_i$ unchanged. An input in $\mathcal{Z}_i$ remains in $\mathcal{Z}_i$, and an input outside the set remains outside it. Since $\mathcal{Z}_i \oplus d \oplus d' = \mathcal{Z}_i$ whenever d and d' both preserve $\mathcal{Z}_i$, and $0 \in \mathcal{D}(\mathcal{Z}_i)$ always, $\mathcal{D}(\mathcal{Z}_i)$ is a subgroup of $(\{0,1\}^n, \oplus)$. Thus $|\mathcal{D}(\mathcal{Z}_i)|$ is a power of two.

Proposition 1. *Fix an instruction i and an n-bit subkey value k, and let $d \in \mathcal{D}(\mathcal{Z}_i)$. Let v denote the known cell value in the targeted lane before subkey addition, so that its S-box input is $v \oplus k$. With the other subkeys fixed, for every value of v the oracle outputs a ciphertext under subkey value k exactly when it outputs one under $k \oplus d$. Consequently an attacker targeting only instruction i cannot determine k more precisely than its coset $k \oplus \mathcal{D}(\mathcal{Z}_i)$, and recovers at most $n - \log_2 |\mathcal{D}(\mathcal{Z}_i)|$ bits of it.*

Proof. By the constraint in Section 3.2, the lane is ineffective under subkey value k exactly when $v \oplus k \in \mathcal{Z}_i$. As d preserves $\mathcal{Z}_i$, we have $v \oplus k \in \mathcal{Z}_i$ if and only if $v \oplus k \oplus d \in \mathcal{Z}_i$, so the two subkey values give the same ineffectiveness decision in that lane for every v. The other lanes are unchanged, so the oracle gives the same one-bit response on every query. No observation distinguishes the two values. □

Therefore, an instruction is usable on its own only if $\mathcal{D}(\mathcal{Z}_i) = \{0\}$. A high ineffective probability does not guarantee this condition. In both `GIFT` implementations, the instruction with the highest ineffective probability, $p = 7/8$, has $\mathcal{D}(\mathcal{Z}_i) = \{0, 13\}$, so on its own it leaves two candidates per n-bit subkey and cannot finish the recovery. Specifically,

Table 3: Single-target instructions selected for the five bitsliced implementations. L is the number of instructions in the analysed S-box implementation, candidates are the σ-free instructions whose ineffective set is neither empty nor the whole domain, and usable are the candidates with $\mathcal{D}(\mathcal{Z}_i) = \{0\}$. Index i is the target's position in the analysed instruction sequence, numbered from 1 to L as in Listings 1 and 2. The last column gives its ineffective probability.

Implementation	L	Candidates	Usable	Index i	Operation	p_i
GIFT-64/128	23	17	7	19	`eor r20, r22`	3/4
GIFT-128/128	23	17	7	19	`eor r20, r22`	3/4
SKINNY-64/64	24	15	8	20	`eor r19, r22`	3/4
SKINNY-128/128	48	31	8	37	`mov r26, r19`	3/4
PIPO-64/128	74	61	27	44	`mov r17, r16`	3/4

instruction 17 (`mov r22, r18`) in Listing 1 cannot distinguish subkey values k and $k \oplus 13$. For every known cell value v, the inputs $v \oplus k$ and $v \oplus k \oplus 13$ either both belong to its ineffective set or both lie outside it, so they produce identical answers. In contrast, instruction 19 (`eor r20, r22`) has ineffective probability 3/4 and $\mathcal{D}(\mathcal{Z}_{19}) = \{0\}$, so repeated queries can distinguish all possible n-bit subkey values despite its lower ineffective probability.

3.4 Target Instruction Selection

For a single-target attack, we consider σ-free instructions with a nonempty ineffective set and $\mathcal{D}(\mathcal{Z}_i) = \{0\}$. The first condition avoids dependence on register values left by earlier code. A nonempty set permits ciphertext output, while the condition on set-preserving XOR differences excludes instructions that leave indistinguishable key candidates. An instruction whose ineffective set is the whole input domain provides no information and is also excluded. Consequently, these conditions identify which instructions can support single-target key recovery. Section 4.3 considers combinations that relax the condition on set-preserving XOR differences.

In the known-plaintext attack, a higher ineffective probability reduces the expected p_i^{-m} injections needed to obtain one ciphertext output under uniform, independent inputs. However, key recovery also depends on how many outputs are needed to remove all incorrect subkey candidates. Since only executions that produce a ciphertext update the candidates, their inputs $x = v \oplus k$ are uniform within $\mathcal{Z}_i$, rather than the full S-box input domain. A fixed incorrect candidate $k' = k \oplus d$ survives an output when $x \oplus d$ also belongs to $\mathcal{Z}_i$. Its rejection probability therefore depends on the overlap between $\mathcal{Z}_i$ and $\mathcal{Z}_i \oplus d$, so equal ineffective probabilities need not give equal key-recovery costs. For both plaintext settings, we treat each usable single target as a length-one instruction sequence and use Section 4.4 to calculate its expected injection count or an enclosing interval. We exclude a target if its entire interval lies above that of another usable single target, and compare the remaining candidates in simulation. Overlapping intervals do not establish a unique minimum. Table 3 lists the selected target for each cipher implementation.

Table 3 summarises the five S-box implementations. Of their instructions, those whose ineffective set is empty, or is the whole domain, or depends on σ drop out first, which leaves between 15 and 61 candidates. Of those, the ones with a nonzero set-preserving XOR difference are unusable on their own, leaving between 7 and 27 usable instructions. In every implementation the highest ineffective probability among them is 3/4. The table identifies the instruction used in each single-target attack on the device.

4 Key Recovery

4.1 Key Recovery with a Known Plaintext

To recover the round key K_{r-1}, we target the S-box layer of round r. Using the plaintext and round keys already recovered, the attacker computes $A_{r-1}(Y_{r-1})$, the state immediately before K_{r-1} is XORed. Its cell value v in the targeted lane is known. The actual S-box input $x = v \oplus k$ remains unknown because the corresponding n-bit subkey k from K_{r-1} has not yet been recovered. For `GIFT` and `SKINNY`, the first target is round 2, where Y_1 is the plaintext, and the attack recovers K_1. For `PIPO`, the first target is round 1, where $A_0(Y_0)$ is the plaintext, and the attack recovers the whitening key K_0. After recovering this key, the attacker computes the state before the next round-key addition and then attacks the following S-box layer to recover the next key. The number of round keys required for master-key recovery depends on the cipher. Four of the implementations of Table 2 need two, and `GIFT`-64/128 needs four, because each of its round keys covers only a quarter of the key.

Let i be the index of the target instruction and g the index of the targeted group among the G groups in round r. A query takes a plaintext, injects at the occurrence of op_i that belongs to group g of round r, and records whether a ciphertext came back. Write $v_1, \ldots, v_m$ for the cell values of $A_{r-1}(Y_{r-1})$ in group g, before adding K_{r-1}, and $k_1, \ldots, k_m$ for the corresponding n-bit subkey values of the round key K_{r-1}, which it seeks to recover. Here j indexes an S-box within group g. Its input is $x_j = v_j \oplus k_j$. As in Section 3.2, an output event gives m constraints at once, $k_j \in v_j \oplus \mathcal{Z}_i$ for every j. The attacker keeps a candidate set $\mathcal{K}_j$ for the subkey value k_j, initially all of $\{0,1\}^n$, and replaces it by $\mathcal{K}_j \cap (v_j \oplus \mathcal{Z}_i)$ on each output event. We discard a query without ciphertext output in this procedure because its constraint concerns combinations of subkey values, whereas these updates maintain only an independent candidate set for each n-bit subkey.

For a single target with $\mathcal{D}(\mathcal{Z}_i) = \{0\}$, a group is recovered when each candidate set $\mathcal{K}_j$ contains only the correct subkey value k_j. Let $N_C^{(i,j)}$ denote the number of ciphertext outputs needed to reach this condition for subkey k_j, and let N_{inj} denote the total number of fault injections needed to recover the subkeys from R attacked S-box layers with G groups each. To calculate its expectation, we assume that the observed plaintexts P are independent and uniformly random and that the same target i is used in every group. Since the preceding transformations are invertible, the S-box inputs in each group are also independent and uniformly distributed. Thus,

$$\mathbb{E}[N_{\mathrm{inj}}] \;=\; RGp_i^{-m}\,\mathbb{E}\left[\max_{1\le j\le m} N_C^{(i,j)}\right].$$

The expected cost therefore depends on both the ciphertext output probability and the number of outputs needed to eliminate the incorrect candidates. Algorithm 1 gives the procedure for one group using a sequence $(i_1, \ldots, i_t)$, where t is the number of target instructions in the sequence. The single-target case has $t = 1$. For $t > 1$, the attacker changes the target across separate executions, while still skipping only one instruction per execution. Furthermore, running Algorithm 1 for every group of a round gives the whole of K_{r-1}.

However, the intersection assumes that an output event is always genuine. If a glitch occasionally fails to fire, a correct ciphertext is read as an output event and the intersection can drop the true key. The existing framework [KSK+26] handles this by relaxing the intersection, which applies here unchanged. Section 5.3 measures how often our rig produces such an observation, and the answer is close enough to never that we use the plain intersection throughout.

Algorithm 1 Known-plaintext recovery of one group of K_{r-1}

Require: round r, group g, sequence $(i_1, \ldots, i_t)$, round keys of rounds $< r-1$
Require: $\bigcap_{s=1}^{t} \mathcal{D}(\mathcal{Z}_{i_s}) = \{0\}$
Ensure: the n-bit subkey values $k_1, \ldots, k_m$ for group g

$\mathcal{K}_j \leftarrow \{0,1\}^n$ **for** $j = 1, \ldots, m$
for $s = 1$ **to** t **do**
  aim at the occurrence of op_{i_s} in group g of round r
  $F_s \leftarrow \left|\bigcap_{u \le s} \mathcal{D}(\mathcal{Z}_{i_u})\right|$ ▷ candidates cannot go below this
  while $|\mathcal{K}_j| > F_s$ for some j **do**
    observe the next uniform random plaintext P and query the oracle
    if the device outputs a ciphertext **then**
      compute $A_{r-1}(Y_{r-1})$ from P and the recovered round keys
      $v_1, \ldots, v_m \leftarrow$ its cell values in group g
      $\mathcal{K}_j \leftarrow \mathcal{K}_j \cap (v_j \oplus \mathcal{Z}_{i_s})$ **for** $j = 1, \ldots, m$ ▷ Section 3.2
    end if ▷ discard the query if no ciphertext is output
  end while
end for
return $(\min \mathcal{K}_1, \ldots, \min \mathcal{K}_m)$ ▷ each $\mathcal{K}_j$ is now $\{k_j\}$ if $\bigcap_s \mathcal{D}(\mathcal{Z}_{i_s}) = \{0\}$

4.2 Key Recovery with Chosen Plaintexts

The attack above discards queries without ciphertext output, which account for most queries. Choosing plaintexts lets the attacker vary one cell before the targeted subkey addition while keeping the other lanes ineffective. Both outcomes then directly constrain the corresponding n-bit subkey.

The idea is to test one cell at a time. First find a plaintext that produces a ciphertext when instruction i is skipped in group g. Under uniform, independent inputs, this search requires p_i^{-m} injections in expectation and establishes $v_j \oplus k_j \in \mathcal{Z}_i$ for all m cells. Keep this plaintext as a base. For each subsequent test, choose a new plaintext whose state immediately before adding K_{r-1} agrees with the base in every cell except the j-th cell of group g, where it takes a uniformly random value v'_j. The S-box input in this lane changes from $v_j \oplus k_j$ to $v'_j \oplus k_j$ without requiring knowledge of k_j. The other $m-1$ lanes remain ineffective, so a ciphertext is output exactly when $k_j \in v'_j \oplus \mathcal{Z}_i$. If no ciphertext is output, then $k_j \notin v'_j \oplus \mathcal{Z}_i$. To obtain the new plaintext, the attacker applies A_{r-1}^{-1} to the desired state and then reverses the preceding rounds using the key terms already recovered.

The ciphertext obtained during the base search reduces all m candidate sets by intersecting each $\mathcal{K}_j$ with $v_j \oplus \mathcal{Z}_i$. This update of the candidate sets uses the output already obtained from the base and requires no separate fault injection. Each subsequent cell test uses a new plaintext derived from the base and requires one fault injection. We apply this update before testing individual cells. Each target uses a fresh base, and Algorithm 2 includes the update from its successful ciphertext output.

Each test after the base is found therefore answers one bit about one n-bit subkey instead of costing p_i^{-m} injections for an output event, so the price of an observation drops from p_i^{-m} to one. Against this, the constraints are weaker, one subkey at a time rather than m at once, and the number of these tests grows with the number of n-bit subkeys. The balance is nonetheless strongly in favour of choosing the plaintexts, and Section 5 measures a factor of between 2.6 and 3.3 across the five implementations.

4.3 Combining Several Target Instructions

Proposition 1 rules out an instruction with a nonzero set-preserving XOR difference, and the instruction it rules out is sometimes the cheapest one available. In both `GIFT`

Algorithm 2 Chosen-plaintext recovery of one group of K_{r-1}

Require: round r, group g, sequence $(i_1, \ldots, i_t)$, round keys of rounds $< r-1$
Require: $\bigcap_{s=1}^{t} \mathcal{D}(\mathcal{Z}_{i_s}) = \{0\}$
Ensure: the n-bit subkey values $k_1, \ldots, k_m$ for group g

$\mathcal{K}_j \leftarrow \{0,1\}^n$ **for** $j = 1, \ldots, m$
for $s = 1$ **to** t **do**
  aim at the occurrence of op_{i_s} in group g of round r
  repeat ▷ find a base state: expected $p_{i_s}^{-m}$ injections
    draw a uniformly random N-bit state b
    compute P from b by reversing the operations before K_{r-1} is added
    query the oracle on P
  until the device outputs a ciphertext
  $v_1, \ldots, v_m \leftarrow$ the cell values of b in group g
  $\mathcal{K}_j \leftarrow \mathcal{K}_j \cap (v_j \oplus \mathcal{Z}_{i_s})$ **for** $j = 1, \ldots, m$
  $F_s \leftarrow \left|\bigcap_{u \le s} \mathcal{D}(\mathcal{Z}_{i_u})\right|$
  for $j = 1$ **to** m **do**
    while $|\mathcal{K}_j| > F_s$ **do**
      draw a uniform v'_j and copy b to b'
      replace cell j of group g in b' by v'_j
      compute P' from b' by reversing the operations before K_{r-1} is added
      query the oracle on P'
      if the device outputs a ciphertext **then**
        $\mathcal{K}_j \leftarrow \mathcal{K}_j \cap (v'_j \oplus \mathcal{Z}_{i_s})$
      else
        $\mathcal{K}_j \leftarrow \mathcal{K}_j \setminus (v'_j \oplus \mathcal{Z}_{i_s})$ ▷ the other $m-1$ lanes are still ineffective
      end if
    end while
  end for
end for
return $(\min \mathcal{K}_1, \ldots, \min \mathcal{K}_m)$

implementations the highest ineffective probability is $p = 7/8$, which at $m = 8$ lanes costs $(8/7)^8 \approx 2.9$ injections per output event against $(4/3)^8 \approx 10.0$ for the best usable alternative at $p = 3/4$, and it belongs to an instruction with $\mathcal{D}(\mathcal{Z}_i) = \{0, 13\}$.

However, nothing forces the attacker to spend all its queries on one instruction. Suppose it targets instructions $i_1, \ldots, i_t$ across different queries, each query still skipping exactly one of them. A subkey candidate survives all the constraints collected for i_1 and all those collected for i_2, and so on, so the argument of Proposition 1 now applies to each $\mathcal{D}(\mathcal{Z}_{i_j})$ separately: the subkey is determined up to $\bigcap_{j=1}^{t} \mathcal{D}(\mathcal{Z}_{i_j})$, which equals $\{0\}$ when no nonzero XOR difference preserves every ineffective set. Specifically, the requirement moves from each instruction to the collection, so an instruction with nonzero set-preserving XOR differences becomes usable when the other targets distinguish the subkey candidates it leaves indistinguishable. We call an ordered list of instructions used this way a target *sequence*, written $(i_1, \ldots, i_t)$. To specify when the target changes, let $H_0 = \{0,1\}^n$ and $H_s = \bigcap_{u=1}^{s} \mathcal{D}(\mathcal{Z}_{i_u})$. The attacker uses target i_s until the candidate set for each n-bit subkey reaches $k \oplus H_s$, then switches to i_{s+1}. We call this exhausting the target. The subkey is recovered when $H_t = \{0\}$. This procedure applies to any useful sequence length, and the search below imposes no two-target cutoff.

This does not change the fault model. Each execution is still disturbed once and still skips a single instruction, and the attacker never needs two skips in the same execution. The attacker needs an injection time for each target instruction, found in the same way as

for a single target. The cost of the extra preparation is paid once and does not scale with the number of queries.

Furthermore, a sequence can be cheaper than any single instruction even when a usable single one exists. A first target with nonzero set-preserving XOR differences reduces the candidate sets to cosets of $\mathcal{D}(\mathcal{Z}_{i_1})$, but cannot distinguish the elements within each coset. A second target can resolve these remaining candidates when its set-preserving XOR differences intersect those of the first target only at 0. Splitting the work this way can reduce the query cost compared with a single target. Section 5.3 measures this separately from the single-target attack in both plaintext settings.

4.4 Efficient Target Sequence Selection

We select an efficient target sequence by calculating the expected number of fault injections needed to eliminate incorrect subkey candidates at each step. The attacker switches to the next target only when the current target cannot distinguish the remaining subkey candidates, and uses the same sequence for every group. The calculation follows the uniform, independent sampling of plaintexts in Algorithm 1 and of states and tested cell values in Algorithm 2. For each new target in the chosen-plaintext attack, the attacker finds a base plaintext that produces a ciphertext when that instruction is skipped and uses this output to update all candidate sets. We count all injections during key recovery, including those used to find these base plaintexts. We exclude injections used to find the glitch timing and pulse settings needed to skip the target instructions.

Let n denote the S-box input size in bits. After exhausting the targets used so far, each subkey's candidate set is $k \oplus H$, where H is the intersection of the set-preserving XOR differences of those targets. Exhausting the next target j leaves $k \oplus H'$, with $H' = H \cap \mathcal{D}(\mathcal{Z}_j)$. We omit target j when $H' = H$, since it cannot eliminate any remaining candidate. Otherwise, H' is a proper subgroup of H, so the number of remaining candidates decreases by at least a factor of two. Starting with 2^n candidates, there can therefore be at most n useful targets.

For each pair (H, j), we calculate the expected number of injections needed with target j to reduce the candidates from $k \oplus H$ to $k \oplus H'$. Future inputs are sampled uniformly and independently, and each new chosen-plaintext target uses a fresh base plaintext. Thus, this expected cost depends on H and the next target j, rather than on the earlier targets. If another sequence reaches the same H and then uses j, we reuse the expected injection count already calculated for reducing $k \oplus H$ to $k \oplus H'$ with target j. Earlier targets' costs are included separately in each sequence's total. Furthermore, under uniform inputs, identical ineffective sets and sets related by an XOR translation have the same transition costs and set-preserving differences. We retain one representative of each such class and record the equivalent instruction indices.

For a wrong candidate $k \oplus d$, define the number of ineffective inputs that reject it as

$$b_j(d) = \left|\{x \in \mathcal{Z}_j : x \oplus d \notin \mathcal{Z}_j\}\right|.$$

For a fixed unknown subkey k, a uniformly sampled value v before key addition gives a uniform S-box input $x = v \oplus k$, which remains unknown to the attacker. Conditioned on ciphertext output, x is uniform in $\mathcal{Z}_j$ for a known-plaintext observation. A chosen-plaintext query uses a new value of v chosen uniformly, so its S-box input x is uniform in $\{0,1\}^n$ and either answer is used. Thus their rejection probabilities are, respectively,

$$r_{\mathrm{K}}(d) = \frac{b_j(d)}{|\mathcal{Z}_j|}, \qquad r_{\mathrm{C}}(d) = \frac{2b_j(d)}{2^n}.$$

The factor two in the second expression counts both directions of crossing the boundary of $\mathcal{Z}_j$. Candidates whose differences belong to the same coset of H' always survive or are

rejected together. It is therefore sufficient to choose one representative of each nonzero coset of H/H', forming a set $\mathcal{W}$.

The candidate set reaches $k \oplus H'$ when target j is exhausted. After a single observation, wrong candidates $k \oplus d$ with $d \in H \setminus H'$ can still remain. For $A \subseteq \mathcal{W}$, $s_{\mathrm{K}}(A)$ and $s_{\mathrm{C}}(A)$ are the probabilities that every candidate $k \oplus d$ with $d \in A$ remains after one ciphertext output or one cell test, respectively. Other candidates may be eliminated. Write $f_j(x) = 1$ for $x \in \mathcal{Z}_j$ and zero otherwise. A candidate remains when $f_j(x \oplus d) = f_j(x)$. The numerators below count input values x satisfying this condition for every $d \in A$. If this count is zero, every observation eliminates at least one of the specified candidates.

$$\begin{aligned} s_{\mathrm{K}}(A) &= \frac{|\{x \in \mathcal{Z}_j : x \oplus d \in \mathcal{Z}_j \text{ for all } d \in A\}|}{|\mathcal{Z}_j|}, \\ s_{\mathrm{C}}(A) &= \frac{|\{x \in \{0,1\}^n : f_j(x \oplus d) = f_j(x) \text{ for all } d \in A\}|}{2^n}. \end{aligned} \tag{2}$$

Let $F_{\mathrm{K}}(q)$ be the probability that one subkey's candidate set reaches $k \oplus H'$ after q ciphertext outputs, and $F_{\mathrm{C}}(q)$ the probability that it reaches the same set after the successful base output and q subsequent cell tests. The candidate set reaches $k \oplus H'$ exactly when none of the wrong candidates represented by $\mathcal{W}$ remains. Using the probabilities in (2), inclusion-exclusion calculates this probability as one minus the individual survival probabilities, plus the probabilities that pairs survive together, with alternating signs for larger subsets,

$$\begin{aligned} F_{\mathrm{K}}(q) &= \sum_{A \subseteq \mathcal{W}} (-1)^{|A|} s_{\mathrm{K}}(A)^q, \qquad q \geq 1, \\ F_{\mathrm{C}}(q) &= \sum_{A \subseteq \mathcal{W}} (-1)^{|A|} s_{\mathrm{K}}(A) s_{\mathrm{C}}(A)^q, \qquad q \geq 1, \\ F_{\mathrm{C}}(0) &= \sum_{A \subseteq \mathcal{W}} (-1)^{|A|} s_{\mathrm{K}}(A). \end{aligned} \tag{3}$$

Here $F_{\mathrm{K}}(0) = 0$, since target j has provided no output yet. $F_{\mathrm{C}}(0)$ is the probability that the successful base output alone reduces the candidates to $k \oplus H'$, before any cell test. The term with $A = \varnothing$ in (3) is one.

In the known-plaintext setting, all m independent lanes must finish and each ciphertext output costs p_j^{-m} injections in expectation. In the chosen-plaintext setting, the base search has that cost, followed by m separate n-bit subkey recoveries. Let $c_B(H, j)$ be the expected number of injections needed with target j to reduce all m subkey candidate sets in one group from $k \oplus H$ to $k \oplus H'$. Here $B = \mathrm{K}$ denotes known plaintexts and $B = \mathrm{C}$ denotes chosen plaintexts. Using the tail-sum expression for the expectation gives the group transition costs

$$\begin{aligned} c_{\mathrm{K}}(H, j) &= p_j^{-m} \sum_{q=0}^{\infty} \bigl(1 - F_{\mathrm{K}}(q)^m\bigr), \\ c_{\mathrm{C}}(H, j) &= p_j^{-m} + m \sum_{q=0}^{\infty} \bigl(1 - F_{\mathrm{C}}(q)\bigr). \end{aligned} \tag{4}$$

The expected cost of a sequence is the sum of its transition costs, multiplied by RG for recovery over all attacked rounds and groups,

$$\mathbb{E}[N_{\mathrm{inj}}] = RG \sum_{s=1}^{t} c_B(H_{s-1}, i_s).$$

For example, instruction 17 in the `GIFT` implementations leaves $H = \{0, 13\}$. Instruction 19 has $|\mathcal{Z}_{19}| = 12$ and $b_{19}(13) = 4$, so the remaining candidate is rejected with probability

1/3 per ciphertext output or 1/2 per chosen-plaintext query. For $m = 8$, the expected group cost of this second step is approximately 71.949 injections with known plaintexts and 20.655 with chosen plaintexts, including the latter's fresh base search.

Enumerating every subset in (3) is expensive when $|\mathcal{W}|$ is large. We then calculate an interval containing the expected number of observations needed to reach $k \oplus H'$, using the hardest individual wrong coset and a union bound over all wrong cosets. Let $a_B(d) = 1 - r_B(d)$ be the probability that the wrong candidate $k \oplus d$ survives one ciphertext output ($B = \mathrm{K}$) or one cell test ($B = \mathrm{C}$). The corresponding inequalities for the expected group cost are

$$\begin{aligned} p_j^{-m} \max_{d \in \mathcal{W}} \sum_{q=0}^{\infty} \left[1 - (1 - a_{\mathrm{K}}(d)^q)^m\right] &\le c_{\mathrm{K}}(H, j) \\ &\le p_j^{-m} \sum_{q=0}^{\infty} \min\left\{1, m \sum_{d \in \mathcal{W}} a_{\mathrm{K}}(d)^q\right\}, \end{aligned} \tag{5}$$

and

$$p_j^{-m} + m \max_{d \in \mathcal{W}} \frac{a_{\mathrm{K}}(d)}{r_{\mathrm{C}}(d)} \le c_{\mathrm{C}}(H, j) \le p_j^{-m} + m \sum_{q=0}^{\infty} \min\left\{1, \sum_{d \in \mathcal{W}} a_{\mathrm{K}}(d) a_{\mathrm{C}}(d)^q\right\}. \tag{6}$$

These inequalities give an interval containing the expected injection count over the stated input distributions. They hold mathematically under this model, while the injection count in an individual run may fall outside the resulting interval. All rejection probabilities are positive because $\mathcal{W}$ excludes H'. To evaluate the infinite sums, we retain the first $V \ge 1$ terms, with $q = 0, \ldots, V - 1$. The omitted terms contribute at most

$$\begin{cases} m p_j^{-m} \displaystyle\sum_{d \in \mathcal{W}} \frac{a_{\mathrm{K}}(d)^V}{1 - a_{\mathrm{K}}(d)}, & \text{known plaintexts,} \\ m \displaystyle\sum_{d \in \mathcal{W}} \frac{a_{\mathrm{K}}(d) a_{\mathrm{C}}(d)^V}{1 - a_{\mathrm{C}}(d)}, & \text{chosen plaintexts,} \end{cases} \tag{7}$$

to the expected injection count. These expressions follow by summing the geometric series for each candidate's survival probability from $q = V$ onward. The factor m accounts for the lanes, and p_j^{-m} converts ciphertext outputs to expected injections in the known-plaintext setting. The factor $a_{\mathrm{K}}(d)$ in the chosen-plaintext expression accounts for survival of the successful base output. Thus (7) limits the numerical error from truncating the sums. For small $\mathcal{W}$, we instead use (3), retaining exact input counts and applying the same calculation only to the uncomputed terms.

Fix a plaintext setting B. For a step from H to H' using target j, let $\ell(H, j)$ and $u(H, j)$ denote the smaller and larger values of the calculated interval for its expected injection count, so that $\ell(H, j) \le c_B(H, j) \le u(H, j)$. For a large $\mathcal{W}$, these values are the left and right expressions in (5) or (6), according to B. For a small $\mathcal{W}$, we evaluate (4) with $F_{\mathrm{K}}(q)$ or $F_{\mathrm{C}}(q)$ from (3) for $q = 0, \ldots, V - 1$. Since every omitted term is nonnegative, this truncated sum gives $\ell(H, j)$, and adding (7) to it gives $u(H, j)$. Let $L(H)$ and $U(H)$ similarly specify an interval containing the smallest expected number of injections needed to reach $\{0\}$ from H for one group. For each possible next target j, we add its cost to the best remaining cost from H'. Thus the comparison includes all subsequent targets through $L(H')$ and $U(H')$. Since every useful target strictly reduces H, dynamic programming computes these values from those for smaller subgroups using

$$L(H) = \min_j \big(\ell(H, j) + L(H')\big), \qquad U(H) = \min_j \big(u(H, j) + U(H')\big). \tag{8}$$

The minima range over useful targets from which $\{0\}$ can be reached. When $H = \{0\}$, the subkey is recovered and no further injections are needed, so $L(\{0\}) = U(\{0\}) = 0$. If no sequence reaches $\{0\}$ from H, we set $L(H) = U(H) = \infty$. If $U(H_0)$ is infinite, no available sequence permits full recovery. Otherwise, the recurrence for U also supplies a sequence from H_0 to $\{0\}$ with expected cost at most $U(H_0)$. Let $P(H)$ be the smallest sum of the ℓ values over all target sequences taking H_0 to H. Whichever of these sequences precedes a step from H, its expected cost is at least $P(H)$.

Proposition 2 (Safe sequence pruning)**.** *Let j be a useful target at H, and let $H' = H \cap \mathcal{D}(\mathcal{Z}_j)$. If $P(H) + \ell(H, j) + L(H') > U(H_0)$, then every target sequence that uses j to reduce $k \oplus H$ to $k \oplus H'$ has a larger expected cost than the sequence supplying $U(H_0)$.*

Proof. The expected cost of a sequence is the sum of its step costs, and a step from H with target j satisfies $\ell(H, j) \le c_B(H, j) \le u(H, j)$. We first show that $L(H')$ is at most the expected cost of every sequence taking H' to $\{0\}$, by induction on $|H'|$. For $H' = \{0\}$, both are zero. Otherwise, such a sequence starts with a useful target j' that leads to a smaller subgroup H'', so its expected cost is at least $\ell(H', j') + L(H'')$ by the induction hypothesis, which is at least $L(H')$ by (8). Similarly, following the targets that attain the minima for U in (8) from H_0 gives a sequence whose expected cost is at most $U(H_0)$, since each of its steps (H, j) costs at most $u(H, j)$.

Now consider a sequence that uses j at H. Its targets taking H_0 to H cost at least the sum of their ℓ values, and hence at least $P(H)$. Its step with j costs at least $\ell(H, j)$, and its targets taking H' to $\{0\}$ cost at least $L(H')$. Its expected cost is therefore at least $P(H) + \ell(H, j) + L(H')$, which is larger than $U(H_0)$ and hence larger than the expected cost of the sequence obtained from the minima for U. □

The search proceeds as follows. We first merge identical and XOR-translated ineffective sets. Starting from H_0, we generate every subgroup obtained by intersecting with $\mathcal{D}(\mathcal{Z}_j)$ for a useful target j, and calculate $\ell(H, j)$ and $u(H, j)$ for each such step. Representing each subgroup as a node and each step from H to H' with target j as an edge (H, j, H'), a target sequence is a path from H_0 to $\{0\}$. We then compute L and U by (8) in increasing order of $|H|$, compute P in decreasing order of $|H|$, and remove every edge that satisfies the condition of Proposition 2. Specifically, pruning removes a path only when its entire calculated interval for the expected injection count lies above that of a competing sequence from H_0 to $\{0\}$. A finite-sample mean is not used in this comparison. We round interval endpoints outward and retain paths whose intervals for the expected injection count overlap. When the intervals are narrow enough, this identifies a lowest-cost sequence to the computed precision. Otherwise, it reduces the candidates to be assessed in simulation without asserting their cost order.

5 Experimental Evaluation

5.1 Experimental Setup

The five implementations run on the ATxmega128D4 of a CW308T-XMEGA target, clocked at four times the 7.37 MHz reference, under the redundancy countermeasure of Section 2.1. The firmware encrypts twice, compares the results, and answers a query either with the ciphertext or with a one-byte refusal. Only the S-box implementations are written in AVR assembly, with the same operations in the same order as the C implementations, and the rest of the firmware is C compiled by `avr-gcc` with `-O3`. In the compiled S-boxes, two-cycle loads are interleaved with the bitwise operations, and a glitch aimed after a load skips the load instead of the following operation. The assembly groups the loads at the front, so each bitwise operation is a single-cycle instruction that a glitch can target.

Table 4: Sequence search reduction from ineffective sets. Initial counts exclude identical sets and targets that do not reduce the current subgroup. Merged counts also identify XOR-translated sets. The final two columns count sequences from H_0 to $\{0\}$ retained by deterministic cost pruning in each plaintext setting.

	Initial	Merged	Retained	
Implementation	sequences	sequences	Known	Chosen
`GIFT`-64/128	378	230	5	6
`GIFT`-128/128	378	230	5	6
`SKINNY`-64/64	285	98	4	4
`SKINNY`-128/128	817 595	230 158	128	296
`PIPO`-64/128	1 662 103	614 434	267	403

Faults are injected by clock glitches from a ChipWhisperer-Husky [OC14]. A glitch is specified by its offset within the cycle, its width, the number of consecutive cycles it covers, and how it is combined with the clock. Its firing cycle selects the instruction and is the cycle at which the S-box execution begins for the targeted round and group plus the cycle offset of the instruction, both found as described in Section 5.3. From the disassembly listing, we obtain the instruction sequence, the cycle offsets and the written registers, and compute $\mathcal{Z}_i$, $\mathcal{D}(\mathcal{Z}_i)$ and the dependence on register values left by the preceding code for every instruction. The simulation and the device use this same model, so we compare them trial by trial.

5.2 Results in Simulation

We apply the calculation of Section 4.4 to every usable single target. The usable instructions with ineffective probability 3/4, the highest among them in each implementation, contain every target not excluded by the calculated intervals. There are five in each `GIFT` implementation, six in each `SKINNY` implementation and nine in `PIPO`, and we simulate each of them in 300 trials per plaintext setting. The cost is not determined by p_i and $\mathcal{D}(\mathcal{Z}_i)$. In `SKINNY`-128/128, instructions 37 and 32 both have $p_i = 3/4$ and $\mathcal{D}(\mathcal{Z}_i) = \{0\}$, but cost 1322 and 2760 injections on average with known plaintexts.

For sequences, the search covers every useful length, which is at most three targets for the 4-bit S-boxes and seven for the 8-bit S-boxes. Merging and pruning remove 98.4–99.98 % of the sequences without any simulation (Table 4). For the 4-bit S-boxes, $|\mathcal{W}| \leq 15$, so every transition is calculated by (3) up to the truncation bounded by (7), and all retained sequences, each with two targets, attain the smallest expected cost. For the 8-bit S-boxes, larger transitions use the bounds in (5) and (6), and the intervals of the retained sequences overlap. We therefore simulate the single target and the retained sequences with the smallest right endpoints, which confirms the calculation for the 4-bit S-boxes and compares the remaining candidates for the 8-bit S-boxes. On the device, we run the single target of every implementation and one of the cheapest sequences of each 4-bit S-box.

Table 5 gives the results of 1000 full key-recovery trials per condition. Every simulated single target and sequence recovers the targeted keys in every trial, and the first two trials of each reproduce the cipher-model simulation exactly. On `SKINNY`-128/128 and `PIPO`, no evaluated sequence improves on the single target, including longer ones. For example, with chosen plaintexts, $10 \to 18 \to 26 \to 34$ on `SKINNY`-128/128 costs 500 injections on average against 438 for instruction 37. This does not prove that every unevaluated sequence is more expensive.

Table 5: Injection costs in simulation and by calculation. Single and Multiple are mean injection counts over 1000 full key-recovery trials. Calculated gives the cost of the retained attack, as an expectation for 4-bit S-boxes and an interval containing the expectation for 8-bit S-boxes. These are computed from the ineffective sets, with outward rounding, and are not confidence intervals.

	Known plaintext			Chosen plaintext		
Implementation	Single	Multiple	Calculated	Single	Multiple	Calculated
GIFT-64/128	1962	1284	1287.7	599	499	501.2
GIFT-128/128	1971	1285	1287.7	600	502	501.2
SKINNY-64/64	981	780	791.0	299	252	250.6
SKINNY-128/128	1308	–	[833, 1387]	438	–	[189, 531]
PIPO-64/128	614	–	[307, 650]	210	–	[73, 254]

–: The tested multiple-target sequences required more injections on average than the single target. Multiple-target sequences are $17 \to 19$ (known) and $6 \to 9$ (chosen) for GIFT, and $10 \to 20$ for SKINNY-64/64 in both settings.

Table 6: Single-target key recovery on the device. Counts are mean and median injections over ten trials per condition. All targeted round keys were recovered in every trial.

		Known plaintext		Chosen plaintext	
Implementation	Index i	Mean	Median	Mean	Median
GIFT-64/128	19	1858	1926	586	599
GIFT-128/128	19	2022	2052	610	611
SKINNY-64/64	20	962	976	319	326
SKINNY-128/128	37	1380	1375	430	432
PIPO-64/128	44	550	567	213	220

5.3 Key Recovery on the Device

Before key recovery, we locate the injection time of each target in a profiling phase with a known key. We sweep the injection time over the S-box execution, read the faulty ciphertext of a single encryption through a debug command that bypasses the comparison, and identify the skipped instruction by matching this ciphertext with the predictions of the instruction-level model of Section 5.1. We locate the injection time for each attacked round separately. This preparation is not counted. Table 6 gives single-target recovery with ten trials per condition, and every trial recovered all targeted round keys. Choosing the plaintexts reduces the injection count by a factor of 2.6 to 3.3. In 1000 injections at one target instruction of each implementation, the faulty ciphertexts disagreed with the model only once, on PIPO, so the plain intersection of Section 4.1 suffices on this platform.

A two-target sequence needs one more injection time, which is found from the listing in the same way and lies a few cycles from the first. Table 7 measures the sequences of Table 5 with the same keys as Table 6, and the combinations reduce the mean injection count on all three implementations for which they were selected.

The best sequence depends on the plaintext setting. On GIFT-64/128, $17 \to 19$ costs 1308 injections with known plaintexts against 1633 for $6 \to 9$, since its first target has ineffective probability 7/8. With chosen plaintexts the order reverses, 506 against 642, since a test costs one injection whatever the ineffective probability. On SKINNY-64/64, the first target 10 has the same ineffective probability 3/4 as the single target but four set-preserving XOR differences, so the gain comes from dividing the work alone. Combinations helped on the three implementations with $n = 4$ and did not help on the two with $n = 8$, and we do not claim that the S-box size causes this.

Table 7: Multiple-target key recovery on the device, as mean injections over ten trials per condition. Parenthesised sequences give instruction indices.

	Known plaintext		Chosen plaintext	
Implementation	Single	Multiple	Single	Multiple
GIFT-64/128	1858	1308 (17 → 19)	586	506 (6 → 9)
GIFT-128/128	2022	1289 (17 → 19)	610	511 (6 → 9)
SKINNY-64/64	962	809 (10 → 20)	319	248 (10 → 20)
SKINNY-128/128	1380	–	430	–
PIPO-64/128	550	–	213	–

–: The tested multiple-target sequences required more injections on average than the single target in simulation, so only single-target device results are reported.

6 Conclusion

In this paper, we presented IISFA on bitsliced implementations protected by a redundancy countermeasure. The attack uses ciphertext output to constrain the round key through the ineffective set of a skipped instruction. The number of set-preserving XOR differences gives the number of subkey candidates that remain indistinguishable, and the ineffective probability determines the ciphertext output probability under uniform, independent inputs. Furthermore, using different target instructions across queries distinguishes candidates that a single target cannot separate and reduces the measured injection cost on three of the five implementations we studied. We give key-recovery procedures for both known and chosen plaintexts and evaluate single-instruction attacks and instruction sequences separately. We calculate expected injection counts or enclosing intervals from the ineffective sets, and use subgroup intersections and dynamic programming to reduce the sequence search before simulation. On an 8-bit AVR microcontroller protected by a redundancy countermeasure, every measured key-recovery campaign succeeded in every trial, and the injection counts agreed with the same-seed model run trial for trial.

Use of AI Tools

We used Claude Opus 5.5 (Anthropic) and Codex (OpenAI, GPT-6.1) to run the experiments and to draft and revise the text. All reported calculations and measurements are reproducible from the artifact, and the authors verified the results and take responsibility for the content of this paper.

References

[ANP20] Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. Fixslicing: A new gift representation. *IACR TCHES*, 2020(3):402–427, 2020. https://tches.iacr.org/index.php/TCHES/article/view/8595.

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers. *IACR TCHES*, 2021(1):402–425, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8739.

[BBB+10] Alessandro Barenghi, Guido Bertoni, Luca Breveglieri, Mauro Pellicioli, and Gerardo Pelosi. Low voltage fault attacks to AES and RSA on general purpose processors. Cryptology ePrint Archive, Report 2010/130, 2010. https://eprint.iacr.org/2010/130.

[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, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 124–156. Springer, Heidelberg, December 2021.

[BFP19] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: Optimizing voltage fault injection attacks. *IACR TCHES*, 2019(2):199–224, 2019. https://tches.iacr.org/index.php/TCHES/article/view/7390.

[Bih97] Eli Biham. A fast new DES implementation in software. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 260–272. Springer, Heidelberg, January 1997.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part II*, volume 9815 of *LNCS*, pages 123–153. Springer, Heidelberg, August 2016.

[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 Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 321–345. Springer, Heidelberg, September 2017.

[BS97] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 513–525. Springer, Heidelberg, August 1997.

[Cla07] Christophe Clavier. Secret external encodings do not prevent transient fault analysis. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES 2007*, volume 4727 of *LNCS*, pages 181–194. Springer, Heidelberg, September 2007.

[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 Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 315–342. Springer, Heidelberg, December 2018.

[DEK+18] Christoph Dobraunig, Maria Eichlseder, Thomas Korak, Stefan Mangard, Florian Mendel, and Robert Primas. SIFA: Exploiting ineffective fault inductions on symmetric cryptography. *IACR TCHES*, 2018(3):547–572, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7286.

[ESH+11] Sho Endo, Takeshi Sugawara, Naofumi Homma, Takafumi Aoki, and Akashi Satoh. An on-chip glitchy-clock generator for testing fault injection attacks. *Journal of Cryptographic Engineering*, 1(4):265–270, December 2011.

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

[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, Part VIII*, volume 15491 of *LNCS*, pages 168–203. Springer Nature Singapore, 2024.

[KGAS25] Anup Kumar Kundu, Shibam Ghosh, Aikata Aikata, and Dhiman Saha. ToFA: Towards fault analysis of GIFT and GIFT-like ciphers leveraging truncated impossible differentials. *IACR TCHES*, 2025(3):614–643, 2025. https://eprint.iacr.org/2024/1927.

[KGSR23] Anup Kumar Kundu, Shibam Ghosh, Dhiman Saha, and Mostafizar Rahman. Divide and rule: DiFA - division property based fault attacks on PRESENT and GIFT. In Mehdi Tibouchi and Xiaofeng Wang, editors, *ACNS 23, Part I*, volume 13905 of *LNCS*, pages 89–116. Springer, Heidelberg, June 2023.

[KJK+20] Hangi Kim, Yongjin Jeon, Giyoon Kim, Jongsung Kim, Bo-Yeon Sim, Dong-Guk Han, Hwajeong Seo, Seonggyeom Kim, Seokhie Hong, Jaechul Sung, and Deukjo Hong. PIPO: A lightweight block cipher with efficient higher-order masking software implementations. In Deukjo Hong, editor, *ICISC 20*, volume 12593 of *LNCS*, pages 99–122. Springer, Heidelberg, December 2020.

[KSK+26] Dongwoo Kang, Hanbeom Shin, DongHyeon Kim, Seokhie Hong, and HeeSeok Kim. SDDT: An operation skip attack framework for bitslice ciphers — validated on PIPO. Cryptology ePrint Archive, Report 2026/1368, 2026. https://eprint.iacr.org/2026/1368.

[MHER14] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, September 2014.

[NDE22] Marcel Nageler, Christoph Dobraunig, and Maria Eichlseder. Information-combining differential fault attacks on DEFAULT. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part III*, volume 13277 of *LNCS*, pages 168–191. Springer, Heidelberg, May / June 2022.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *COSADE 2014*, volume 8622 of *LNCS*, pages 243–260. Springer, Heidelberg, April 2014.

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 2–12. Springer, Heidelberg, August 2003.

[SKK+26] Hanbeom Shin, Insung Kim, Sunyeop Kim, Byoungjin Seok, Deukjo Hong, Jaechul Sung, Seokhie Hong, Sangjin Lee, and Dongjae Lee. MIFA: An MILP-based framework for improving differential fault attacks. *IACR TCHES*, 2026(3):465–488, 2026. https://tches.iacr.org/index.php/TCHES/article/view/13101.

[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 TCHES*, 2018(3):150–172, 2018. https://tches.iacr.org/index.php/TCHES/article/view/7272.