Automatic Search of Meet-in-the-Middle Differential Fault Analysis on AES-like Ciphers

Qingyuan Yu\textsuperscript{1,4}, Xiaoyang Dong\textsuperscript{2,6,7,}, Lingyue Qin\textsuperscript{3,6}, Yongze Kang\textsuperscript{1,4}, Keting Jia\textsuperscript{5,6,}, Xiaoyun Wang\textsuperscript{2,4,6} and Guoyan Zhang\textsuperscript{1,4,7}

\textsuperscript{1} School of Cyber Science and Technology, Shandong University, Qingdao, China, \{yuqy, yzkang, guoyanzhang\}@mail.sdu.edu.cn
\textsuperscript{2} Institute for Advanced Study, BNRist, Tsinghua University, Beijing, China, \{xiaoyangdong, xiaoyunwang\}@tsinghua.edu.cn
\textsuperscript{3} BNRist, Tsinghua University, Beijing, China, qinly@tsinghua.edu.cn
\textsuperscript{4} Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Jinan, China
\textsuperscript{5} Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University, Beijing, China, ktjia@tsinghua.edu.cn
\textsuperscript{6} Zhongguancun Laboratory, Beijing, China
\textsuperscript{7} Shandong Institute of Blockchain, Jinan, China

Abstract. Fault analysis is a powerful technique to retrieve secret keys by exploiting side-channel information. Differential fault analysis (DFA) is one of the most powerful threats utilizing differential information between correct and faulty ciphertexts and can recover keys for symmetric-key cryptosystems efficiently. Since DFA usually targets the first or last few rounds of the block ciphers, some countermeasures against DFA only protect the first and last few rounds for efficiency. Therefore, to explore how many rounds DFA can affect is very important to make sure how many rounds to protect in practice. At CHES 2011, Derbez et al. proposed an improved DFA on AES based on MitM approach, which covers one more round than previous DFAs. To perform good (or optimal) MitM DFA on block ciphers, the good (or optimal) attack configurations should be identified, such as the location where the faults inject, the matching point with differential relationship, and the two independent computation paths where two independent subsets of the key are involved. In this paper, we formulate the essential ideas of the construction of the attack, and translate the problem of searching for the best MitM DFA into optimization problems under constraints in Mixed-Integer-Linear-Programming (MILP) models. With the models, we achieve more powerful and practical DFA attacks on SKINNY, CRAFT, QARMA, PRINCE, PRINCEv2, and MIDORI with faults injected in 1 to 9 earlier rounds than the best previous DFAs.

Keywords: Differential fault analysis · Meet-in-the-middle · Automatic search

1 Introduction

Fault analysis (FA), first introduced by Boneh et al. [BDL97] against RSA-CRT implementations, is one of the most powerful attacks on implementations of cryptographic primitives. It allows the attacker to get additional side-channel information and achieve the key recovery attacks in practical time. At CRYPTO 1997, Biham and Shamir [BS97] proposed the differential fault analysis (DFA) on DES block cipher. Since then, various fault attacks were
introduced, such as (statistical) ineffective fault analysis [Cla07, DEK+S18, DEG+S18], collision fault analysis [BK06, Hem04], statistical fault attacks [FJLT13, DEK+S16], fault intensity analysis [LSG+S10], persistent fault attack [ZLZ+S18], fault template attack [SBR+20], fault correlation analysis [SMC21], and other fault attacks [JSC+S14, DPDC+S15, BHW+S18, TLG+S15, SHS16, HS13, GKP+18, MGV+08, PAM+19, RLK+11]. For fault analysis on AES, the fault can be induced in the round counter [CT05, AK97] to reduce the number of rounds, or in the internal state during a round [PQ03, DLV03, BS03, Gir04] to exploit the confusion and diffusion characteristics of the fault, or in the key schedule [CY03, Gir04, TFSY07, DV12]. There are several fault models in [RSG21]. A less common model is the random bit fault model [BS03], which allows the adversary to flip one particular bit. A more common model is the random byte fault model [PQ21].

**Differential Fault Attack [BS97]** exploits the difference between correct and faulty ciphertexts by injecting a fault in the internal state of the last several rounds. With the correct and faulty ciphertext pairs, the adversary can retrieve the secret key by a cryptographic analysis. The key point of DFA is that it allows the adversary to analyse a small number of rounds of a block cipher. DFA has been widely applied to the attacks on DES [Hem04, Riv09], AES [PQ03, DLV03, BS03, Gir04, MSS06, DFL11, YSW18, SHS16], PRESENT [SGSS14, GYS15, PBMB17] and others [FT10, TBM14]. The countermeasures against DFA include cipher or mode level (e.g. FRIET [SBD+20], CRAFT [BLMR19], DEFAULT [BBB+21] and others [MSGR10, MP+11, DFM+15]) and implementation level ways [LRT12], etc. A widely used implementation level countermeasure against DFA is to perform the computation twice and check whether the same result is obtained [MSY06, ML08, JMR07, BBK+10].

Since DFA against block ciphers usually targets the last few rounds, one does not need to protect the whole cipher thus saving computation time [MSY06, CFGR10, BBK+10]. However, the number of rounds to protect must be chosen carefully in order to prevent security flaws. In order to determine how many rounds to protect, one has to know how many rounds DFA can work on. At CHES 2009, Rivain et al. [Riv09] studied against DES by introducing some faults at the end of round $R - 7$, $R - 6$, $R - 5$ or $R - 4$. For AES, most DFA techniques [PQ03, Gir04, Muk09] work by inducing faults at the end of round $R - 3$, $R - 2$ or $R - 1$. Therefore, protecting the last and the first three rounds of AES against DFA is usually suggested [CFGR10]. At CHES 2011, Derbez, Fouque, and Leresteux [DFL11] introduced the Meet-in-the-Middle (MitM) and impossible differential fault analysis on AES by inducing faults at the end of round $R - 4$ for the first time, that broke those implementations only protecting the last three rounds.

**The Meet-in-the-Middle (MitM) approach** is a time-memory trade-off cryptanalysis technique, which can be traced back to Diffie and Hellman’s attack on DES [DH77]. The basic idea is to divide the key space into two independent subsets (also known as neutral sets), and then find matches from the two subsets. Let $E_K(\cdot)$ be a block cipher with block size $n$-bit, such that $C = E_K(P) = F_{K_2}(F_{K_1}(P))$, where $K = K_1 \parallel K_2$ has $n$ bits, and $K_1$ and $K_2$ are neutral key materials of $n/2$ bits. For a given plaintext-ciphertext pair $(P, C)$, a naive exhaust search attack needs a time complexity $2^n$ to find the key. However, the MitM attack computes independently $F_{K_1}(P)$ and $F_{K_2}^{-1}(C)$ with independent guesses of $K_1$ and $K_2$, and searches collision between $F_{K_1}(P)$ and $F_{K_2}^{-1}(C)$ to find the $K$ with a time complexity about $2^{n/2}$. This approach has been widely applied in many cryptanalysis methods, like preimage attacks on hash functions [SA09] and key recovery attacks on block ciphers [BR10]. Derbez et al. [DFL11] for the first time combined the MitM approach with DFA by utilizing the pairwise matching in the differential internal state getting from correct and faulty ciphertexts with different subkey guessing. In the last decades, automatic search has boosted cryptanalysis, e.g. [BDF11, DF16]. Using automatic tools
for solving problems like Mixed Integer Linear Programming (MILP), Satisfiability (SAT), Satisfiability Modulo Theories (SMT) problems, or Constrained Programming (CP), better cryptanalytic results have been achieved in topics including differential/linear attacks [MWGP11, SHW+14, KLT15], impossible differential attacks [ST17, CCF+21, SGL+17], cube or integral attacks based on division properties [XZBL16, TIHM17].

MILP is a method frequently used in business and economics to solve optimization problems. It deals with the problems of optimizing a linear objective function $f(x_1, x_2, \ldots, x_n)$ subject to linear inequalities involving variables $x_i$, $1 \leq i \leq n$. The first attempt to apply the MILP model to cryptanalysis is to determine the minimum number of differential active S-boxes for AES by Mouha et al. [MWGP11]. They assigned Boolean variable $x_i$ for each S-box, where $x_i = 1$ means the $i$-th S-box is active. The variables are restricted by linear inequalities, which are derived from the differential propagation properties of each operation of AES. For example, given XOR operation $x_i \oplus x_j = x_k$, if $x_k = 1$ (active), there is at least one active S-box for the $i$-th and $j$-th S-box, i.e., $x_i + x_j \geq 1$. At EUROCRYPT 2021, Bao et al. [BDG+21] introduced the MILP-based automatic tools for MitM preimage attacks on AES-like hashing, and many further improved tools for MitM have been proposed [DHS+21, BGST22, SS22]. At CRYPTO 2011, Bouillaguet, Derbez, and Fouque [BDF11] proposed an ad-hoc automatic tool to search DFAs on round-reduced AES. Their tool mainly used C++ programs to exhaust all possible DFA attacks on AES by exploiting its details and properties. Applying their tool to other block ciphers does not seem to be trivial, and many recent DFAs on other block ciphers are still done manually [CZS16, KAKS22, VSBM20]. In this paper, we manage to propose an MILP modelling method for the MitM DFAs, which is general for some popular block ciphers.

**Our contributions.** In this paper, we introduce an automatic search model based on MILP for the good or optimal MitM differential fault analysis, which is then successfully applied to several popular lightweight block ciphers. Under the fundamental assumptions of DFA, we can inject a random byte (nibble) fault to the internal state of the block cipher, and the location of the fault in the state could be controlled. We can obtain both correct and faulty ciphertexts. Under the assumptions, we generalize the MitM differential fault analysis, and translate the problems into MILP models by modelling the location and propagation of the differential faults, the matching point with differential relationship, the propagation of the two neutral sets, and the objective functions, etc.

Since many lightweight block ciphers adopt non-MDS (non-Maximum-Distance-Separable Matrix) layer (e.g. SKINNY [BJK+16]), we develop the matching rules for non-MDS operations, which include the modellings for the differential equations and the propagation of the two neutral sets. We develop the propagation rules of the two neutral sets, which will dominate the overall time complexity and should be balanced and minimized. When optimizing the computational complexity of the DFA attacks, the number of faults we need to inject is also one of the parameters to evaluate the strength of the attacks, which is one of the factors we considered in our objective function. To keep our DFA practical, we perform multiple MitM DFAs to recover the full key, while each MitM DFA only recovers a fraction of the key bits. Therefore, we develop the objective function to balance and minimize the time complexities of different MitM DFAs. Our tool is generic by writing down the linear inequalities on the matching part, the propagation of differential faults, the propagation of neutral sets, and the objective function. So, implementing the model for a different block cipher is just to replace the linear inequalities for each part.

As applications, we show better DFAs for SKINNY, CRAFT, QARMA, PRINCE, PRINCEv2, and MIDORI found by our tools in Sect. 4, where the positions of fault injections can be in 1 to 9 earlier rounds than the best previous attacks. We also use our model to search for DFAs on AES-128/-192/-256, but only get improvements to the secondary steps of Derbez et al.’s attacks [DFL11] on AES-192 and AES-256. In [DFL11], Derbez et al. extended
the DFA attacks to AES-192 and AES-256. In the first step, they use MitM approach to recover the subkey in the last round, and in the second step they recover the remaining key bytes following the idea of Piret and Quisquater [PQ03]. The time complexity of the second step is about $2^{40}$. In Section 4.6, we give an improvement on the second step, instead of employing Piret and Quisquater’s method, we use another step of MitM attack. The time complexity of our improved step is $2^{10.6}$. However, the overall time complexity of the DFA is dominated by the first step, so the time complexity of our attack is the same as Derbez et al.’s. We summarise our improved differential fault attacks in Table 1. The attacked “Round” means the number of non-linear layers between the position of the fault injection and the ciphertext. More rounds attacked means that there are more rounds need to be protected when implementing the cipher on devices. Columns with “Round (Previous)” and “Round (Ours)” list the best results of previous DFAs and our improved ones. The “Time” column and “Memory” column show the time complexities and memory complexities of our attacks, and the “# of faults” list the number of byte (nibble) faults need to inject in our attacks. Our model and source codes are publicly available at https://github.com/yqy-yu/MITM-DFA.

Comparisons with previous MILP-based MitM attacks. In mathematic cryptanalysis, MILP-based MitM automatic models [BDG +21, DHS +21, SS22] have been proposed. In their models, they do not need to model the differential fault propagation. Therefore, their matching point is easier and only related to the propagations of two neutral sets, while our modelling for the matching point has to consider the propagations of both the neutral sets and the differential fault. Since our MitM DFA is for the practical attack, the sizes of two neutral sets are kept small, i.e., one MitM attack only recovers a small fraction of the full key. To recover the full key, we perform multiple MitM DFAs. Therefore, the overall time complexity is dominated by an MitM attack with the highest complexity. While previous models [BDG +21, DHS +21, SS22] only consider one MitM attack. Therefore, the objective function is different.

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Round(Previous)</th>
<th>Round(Ours)</th>
<th>Time</th>
<th>Memory</th>
<th># of faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKINNY-128-128/256</td>
<td>5 [VBSM18]</td>
<td>9</td>
<td>$2^{24}$</td>
<td>$2^{24}$</td>
<td>9</td>
</tr>
<tr>
<td>SKINNY-64-64/128</td>
<td>5 [VBSM18]</td>
<td>10</td>
<td>$2^{24}$</td>
<td>$2^{24}$</td>
<td>10</td>
</tr>
<tr>
<td>CRAFT</td>
<td>1 [RVB22]</td>
<td>10</td>
<td>$2^{30}$</td>
<td>$2^{12}$</td>
<td>28</td>
</tr>
<tr>
<td>QARMA-64</td>
<td>-</td>
<td>5</td>
<td>$2^{16}$</td>
<td>$2^{16}$</td>
<td>28</td>
</tr>
<tr>
<td>QARMA-128</td>
<td>-</td>
<td>4</td>
<td>$2^{32}$</td>
<td>$2^{32}$</td>
<td>10</td>
</tr>
<tr>
<td>PRINCE/PRINCEv2</td>
<td>4 [KAKS22]</td>
<td>4</td>
<td>$2^{20}$</td>
<td>$2^{20}$</td>
<td>10</td>
</tr>
<tr>
<td>MIDORI-64</td>
<td>3 [CZS16]</td>
<td>5</td>
<td>$2^{16}$</td>
<td>$2^{16}$</td>
<td>15</td>
</tr>
<tr>
<td>MIDORI-128</td>
<td>3 [CZS16]</td>
<td>5</td>
<td>$2^{24}$</td>
<td>$2^{24}$</td>
<td>9</td>
</tr>
</tbody>
</table>

1 The ability in [KAKS22] is stronger, where they need to inject one-bit faults and know which bit is injected.

2 Preliminaries

2.1 Differential Fault Analysis using MitM

At CHES 2011, Derbez et al. [DFL11] proposed a differential fault analysis on AES using the MitM method shown in Figure 1. Note that $R = 10$ for AES-128. They induced a
random byte fault injection at the end of round \( R - 4 \). For AES-128, according to the differential propagation rules, they can obtain several differential equations of the state at the beginning of Round 9 (i.e., \( X_9 \)) between the correct and faulty states, i.e.,

\[
X_9[0] \oplus \tilde{X}_9[0] = I, \ X_9[1] \oplus \tilde{X}_9[1] = I, \ X_9[2] \oplus \tilde{X}_9[2] = 3I, \ X_9[3] \oplus \tilde{X}_9[3] = 2I,
\]

where \( X_9[i] \) denotes the \( i \)-th byte in the correct state \( X_9 \), \( \tilde{X}_9[i] \) denotes the \( i \)-th byte in the faulty state \( \tilde{X}_9 \), and \( I \in \mathbb{F}_2^8 \). These differential equations can provide a filter of subkey bits by constructing relationships, e.g.,

\[
X_9[0] \oplus \tilde{X}_9[0] = X_9[1] \oplus \tilde{X}_9[1].
\]

Use \( C \) to denote the ciphertext state, the state \( X_9 \) can be represented as

\[
X_9 = \text{SB}^{-1}(\text{SR}^{-1}(\text{MC}^{-1}(\text{ARK}^{-1}(\text{SB}^{-1}(\text{SR}^{-1}(\text{ARK}^{-1}(C))))))).
\]

Denote \( K_0 \) as the whitening key, \( K_i \ (i = 1, 2, ..., 10) \) as the subkey of Round \( i \) and \( U_i = \text{MC}^{-1}(K_i) \). When computing \( X_9[0] \) with Equ. (3), 5 key bytes, i.e., \( K_{10}[0, 7, 10, 13] \) and \( U_{0}[0] \), need to be guessed. Similarly, to compute \( X_9[1] \), another 5 key bytes also need to be guessed, i.e., \( K_{10}[3, 6, 9, 12] \) and \( U_{0}[13] \). Since \( X_9[0] \) and \( X_9[1] \) are computed independently with two independent key subsets, we denote the two key subsets as neutral key sets. For a given pair of correct and faulty ciphertexts, the relationship in Equ. (2) has to be satisfied, which acts as a filter of \( 2^{-8} \) for the subkey space of 10 bytes involved in Equ. (2). In order to identify the one right 10-byte subkey, one has to collect 10 pairs of correct and faulty ciphertexts, which will provide a filter of \( 2^{-8 \times 10} = 2^{-80} \). With the MitM approach, one guesses 5 subkey bytes \( (K_{10}[0, 7, 10, 13], U_{0}[0]) \) to compute the 10-byte of \( X_9[0] \oplus \tilde{X}_9[0] \) of the 10 pairs for matching and store the guessed subkeys in a hash table indexed by the 10 matching bytes. Then, for each guess of \( K_{10}[3, 6, 9, 12] \) and \( U_{0}[13] \), one computes \( X_9[1] \oplus \tilde{X}_9[1] \) for the 10 pairs and checks against the hash table to find a match. One expects 1 match, which should be the correct 10-byte subkey. The time and memory costs are both \( 2^{40} \).

The key point of the MitM DFA is to exploit differential equations, e.g. Equ. (2), whose expressions on both sides depend on different subsets of keys. At CRYPTO 2011, Bouillaguet, Derbez, and Fouque introduced an ad-hoc and dedicated automatic tool to
search MitM DFAs on AES [BDF11]. In this paper, we try to build generic automatic models based on MILP for MitM DFAs to realize the optimal fault injection and key recovery phase.

Generalizations and Notations of MitM DFA. Use Round 1 to Round $R$ to denote the iterated round function of block ciphers, and assume we can inject random cell (byte or nibble) faults at a specified internal state to obtain correct and faulty ciphertexts. Denote the round we inject faults as Round $R_s$, and denote the round $R_m$ as the matching point if there exist differential equations that can act as a filter of subkeys, e.g., $\Delta S[i] = \Delta S[j]$, where $\Delta S[i]$ is only determined by key cells in the neutral set $Set_1$ and $\Delta S[j]$ is determined by neutral set $Set_2$. Thereafter, a MitM DFA is performed to recover the subkey. Figure 2 gives a high-level framework.

2.2 Description of SKINNY

We will take the MILP-based automatic model of MitM DFA attack on SKINNY as an example. Therefore, we briefly describe SKINNY here. SKINNY is a family of lightweight block ciphers designed by Beierle et al. [BJK+16] based on the TWEAKEY framework [JNP14]. The block size is $n = 64$ or 128. There are versions with tweakey size $t = n$-bit ($TK_1$, $TK_2$), or 3$n$-bit ($TK_1,TK_2,TK_3$). The $r$-th ($1 \leq r \leq R$) round function shown in Figure 3 consists of five operations: SubCells (SC), AddConstants (AC), AddRoundTweakey (ART), ShiftRows (SR) and MixColumns (MC).

For AddRoundTweakey (ART), the first and second rows of subtweakey $STK$ are xored to the internal state. Initially, $STK = TK_1$ for SKINNY-$64$-$n$, $STK = TK_1 \oplus TK_2$ when $t = 2n$, and $STK = TK_1 \oplus TK_2 \oplus TK_3$ for $t = 3n$ version. The $STK$ for each round is obtained by tweakey update function, which consists of two parts: firstly, a permutation $P_r = [9,15,8,13,10,14,12,11,0,1,2,3,4,5,6,7]$ is applied to the tweakey arrays. Then, use different LFSRs to update the the first and second rows of $TK_2$ and $TK_3$ if $t = 2n$ or $t = 3n$.

Figure 3: Round function of SKINNY.
3 Program the MitM-DFA with MILP

3.1 Formulate the MitM Differential Fault Analysis

In this section, we show the specific modelling schemes for SKINNY. To facilitate the visualization of our analysis, each cell can take one of the five colors (White, Gray, Red, Blue, and Orange) according to certain rules, and a valid coloring scheme in our model corresponds to a configuration of the MitM differential fault analysis. The semantics of the colors are listed as follows:

- White(W): Inactive in forward differential propagation.
- Gray(G): Active in forward differential propagation.
- Blue(B): Known cells by guessing the key cells in Set1.
- Red(R): Known cells by guessing the key cells in Set2.
- Orange(O): Cells determined by Set1 ∩ Set2.

For SKINNY with $R$ rounds, we set the round of fault injection as $R_s$, and use $R_m$ to denote the round of the matching point. For each internal state $A$ in the rounds $R_s$ to $R_m - 1$, we introduce the Binary variable $a_i^A$ to denote if the $i$-th cell of state $A$ is active, i.e., $a_i^A = 1$ means active cell with Gray color; $a_i^A = 0$ means inactive cell with White.

For the states in the rounds $R_m$ to $R$, we introduce two additional Binary variables $b_i^A$ and $c_i^A$, such that $(a_i^A, b_i^A, c_i^A) = (1, 1, 0)$ represents Blue, $(a_i^A, b_i^A, c_i^A) = (1, 0, 1)$ represents Red, $(a_i^A, b_i^A, c_i^A) = (1, 1, 1)$ represents Orange, $(a_i^A, b_i^A, c_i^A) = (1, 0, 0)$ represents Gray and $(a_i^A, b_i^A, c_i^A) = (0, 0, 0)$ represents White. For rounds from $R_s$ to $R$, the propagation of Gray and White cells depends on the differential propagation. For rounds from $R$ to $R_m$, the propagation of R, B and O rely on the backward computation rules.

3.2 Programming the MILP Model

In this section, we show how to build constraints for each component and how to solve the model for specific block ciphers. Use SKINNY as an example, we show the details below.

3.2.1 Constraints for the states from Round $R_s$ to Round $R_m - 1$

According to the differential property of SKINNY, we model the constraints of active cells before and after SC, SR and MC operations. Use $X_r$, $Y_r$ and $Z_r$ to denote the state before SC, SR and MC operations in round $r$, respectively. Let $STK_r$ denote the tweak key state used in round $r$. Since ARK and AC operations don’t affect the propagation of the differential, we only list the the rules for SC, SR and MC operations below.

- SC: Because we only consider the cell-level truncated differential propagation, for each S-box, the output difference is nonzero if and only if the input difference is nonzero. The constraint for SC is $a_i^{X_r} - a_i^{X_r} = 0, \forall 0 \leq i \leq 15$. We add the constraint $\sum_i a_i^{X_r} = 1$ to make sure that random fault is injected to one cell at the starting point.

- SR: For the SR operation, a cell permutation $P$ is applied, i.e., $a_i^{Z_r} - a_i^{P[r]} = 0, \forall 0 \leq i \leq 15$, where $P = [0, 1, 2, 3, 7, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12]$. 

- MC: For each column $j$ of MC, we build constraints according to the following rules, where $0 \leq j \leq 3$:

$$ a_{X_r+1}^{X_r+1} = a_j^{Z_r} \lor a_{8+4+j}^{Z_r} \lor a_{12+4+j}^{Z_r}, \quad a_{X_r+1}^{X_r+1} = a_j^{Z_r} \lor a_{8+4+j}^{Z_r} \lor a_{12+4+j}^{Z_r}, \quad a_{X_r+1}^{X_r+1} = a_j^{Z_r} \lor a_{8+4+j}^{Z_r} \lor a_{12+4+j}^{Z_r}. $$

Following the method by [SHW+14], we convert the above equations to inequalities.
For the matching point $X_{R_m} = MC(Z_{R_m-1})$, the possible matching rules are given as follows (also in Figure 4):

- **Rule-1**: With input truncated differential form 1*00, the 0-th cell, the 1-th cell, and the 3-th cell of the output difference are possible for matching. For example, for $MC(A) = B$ in Figure 4, since $\Delta A[2] = \Delta A[3] = 0$, $\Delta B[0] = \Delta B[1] = \Delta B[3]$ holds according to the property of $MC$ of SKINNY. Therefore, if $\Delta B[0]$ can be computed by key subset marked by Blue and $\Delta B[3]$ can be computed by another key subset marked by Red, then a MitM attack can be performed with these two neutral key subsets and the relation $\Delta B[0] = \Delta B[3]$, i.e., a matching point exists. We restrict that only the three cells $B[0, 1, 3]$ can be marked by Blue or Red. In other words, if this column has both Blue and Red cells, there exists one matching equation.

- **Rule-2**: With input truncated differential form 1*10, the 0-th cell and the 3-th cell of the output difference are possible for matching.

- **Rule 3**: With input truncated differential form 1*01, the 1-th cell and the 3-th cell of the output difference are possible for matching.

- **Rule-4**: With input truncated differential form 0010, the 0-th cell, the 2-th cell and the 3-th cell of the output difference are possible for matching.

- **Rule-5**: With input truncated differential form 0110, the 0-th cell and the 3-th cell of the output difference are possible for matching.

- **Rule-6**: With input truncated differential form 0011, the 2-th cell and the 3-th cell of the output difference are possible for matching.

We introduce Binary variables $match_i (0 \leq i \leq 3)$ for each column. If there exists at least one Blue cell and one Red cell in the $i$-th column, then $match_i = 1$, otherwise $match_i = 0$ or the model is infeasible. This can be done with the constraints

$$match_i \geq b_{i,j+1}, \quad match_i \geq c_{i,j+1}, \quad match_i \leq \sum_j b_{i,j+1}, \quad match_i \leq \sum_j c_{i,j+1}, \quad \text{for} \quad 0 \leq j \leq 3.$$ 

We call $DoM = \sum_i match_i$ the degree of match, and add constraint $DoM = 1$ to make sure that one differential equation is utilized to act as a filter in each injection phase.

### 3.2.2 Constraints for the states from Round $R$ to Round $R_m$

In this section, we show the constraints for the known values by guessing two neutral key subsets in DFA on SKINNY. Use $b_i^A$ and $c_i^A$ to denote the value of the $i$-th cell in the state $A$, which is known by guessing subkey cells in Set$_1$ and Set$_2$ respectively. The propagation rules of $b_i^A$ and $c_i^A$ are determined by operations in the decryption direction. We list the rules for different operations for $b_i^A$, and the constraints for $c_i^A$ are the same.
We introduce the auxiliary variables $b_i^{Y_r} - b_i^{Y_r} = 0, \forall 0 \leq i \leq 15$. 

**ARK:** For SKINNY, the first and second rows of subtweakey array are XORed to the internal state. If a cell is XORed with the subtweakey state, we can deduce the value of the cell by guessing the corresponding cell in the subtweakey state, else we need not guess any subtweakey cell. Thus we have the following constraints for ARK:

$$b_i^{STK_r} - b_i^{Y_r} = 0 \; (0 \leq i \leq 7), \; b_i^{STK_r} = 0 \; (8 \leq i \leq 15).$$

**SR:** For SR operation, a cell permutation $P_r$ is applied on the internal state, i.e., $b_i^{Y_r} - b_i^{Z_r} = 0, \forall 0 \leq i \leq 15$. 

**MC:** For each column $j$, the known cells in the input column follow the rules below, where $0 \leq j \leq 3$:

$$b_j^{Z_r} = b_{4+j}^{X_{r+1}}, \; b_{4+j}^{Z_r} = b_{4+j}^{X_{r+1}} \land b_{8+j}^{X_{r+1}} \land b_{12+j}^{X_{r+1}}, \; b_j^{Z_r} = b_{4+j}^{X_{r+1}} \land b_{12+j}^{X_{r+1}}, \; b_{12+j}^{Z_r} = b_{j}^{X_{r+1}} \land b_{12+j}^{X_{r+1}}.$$

Following the method by [SHW+14], we convert the equations to inequalities.

According to these principles, we build constraints for internal states from round $R$ to $R_m$.

### 3.2.3 Objective Function and Solving Process

To properly evaluate the performance of our attacks, we take into account two factors: the number of faults required to inject and the time complexity of recovering the key cells for each injection. Since multiple fault injections may be necessary to fully recover the key, the solving phase is a multi-step process.

For SKINNY-n-n, the key schedule is linear, where a cell-wise permutation $P_r$ is applied in each round to update the tweakey matrix. Denote the master tweakey state as $tk_i$, where $tk_i$ represents the $i$-th cell of the state. To prevent repeatedly guessing one byte of the tweakey across multiple rounds, we introduce $KB_i$ and $KR_i (0 \leq i \leq 15)$ to indicate whether each tweakey cell belongs to $Set_1$ or $Set_2$. We constrain the variables with

$$KB_i = \bigvee_{r=R_m}^{R} b_i^{STK_r}_{P_r^{r-1}[i]}, \; \text{and} \; KR_i = \bigvee_{r=R_m}^{R} b_i^{STK_r}_{P_r^{r-1}[i]}.$$

We introduce the auxiliary variables $KG_i (0 \leq i \leq 15)$ to represent whether $tk_i$ has been guessed or not, which are initialized as $KG_i = 0, \forall 0 \leq i \leq 15$. After each fault injection and key recovery phase, we update the state with $KG_i = 1$ if $tk_i$ has been obtained in the key recovery phase, i.e., we update the model according to the following equation after each solving phase:

$$KG_i = KG_i \lor KB_i \lor KR_i, \; \forall 0 \leq i \leq 15.$$

Use $\overline{V}$ to denote the "NOT" operation for any variable $V$. In each solving phase, $n_1 = \sum_{i=0}^{15}(KB_i \land \overline{KG_i})$ and $n_2 = \sum_{i=0}^{15}(KR_i \land \overline{KG_i})$ denote the number of cells that we need to guess in each tweakey subset ($Set_1$ or $Set_2$), and $n_3 = \sum_{i=0}^{15}(KG_i \land \overline{KB_i} \land \overline{KR_i})$ denotes the number of cells in tweakey state that we have not guessed. For each recovery phase, the time complexity to recover the full tweakey state is about $(2^{c_{n_1}} + 2^{c_{n_2}}) \times 2^{c_{n_3}}$. So we set the objective function as

$$\text{Minimize} \left( \sum_{i=0}^{15}(KB_i \land \overline{KG_i}) \lor \sum_{i=0}^{15}(KR_i \land \overline{KG_i}) \lor \sum_{i=0}^{15}(KB_i \land \overline{KG_i}) \right) \lor \text{Max} \left( \sum_{i=0}^{15}(KB_i \land \overline{KG_i}), \sum_{i=0}^{15}(KR_i \land \overline{KG_i}) \right).$$
to search for the attack with the optimal time complexity. To ensure our attack can run in a practical time, we add constraints for $KB_i$ and $KR_i$. For example, we add constraints

$$\text{Max } \left\{ \sum_{i=0}^{15} KB_i, \sum_{i=0}^{15} KR_i \right\} \leq 5$$

to make sure the time complexity of each filter phase is bounded by $2^{40}$.

To perform the key recovery attack with fewer fault injections, we expect to exhaust the matching degree that can be got from each injection. Thus, we set the cell with a fault injection at the starting point to always have a difference, and repeatedly solve the model until there are no differential equations that can be used for matching. That is, if we get $a_i^{X_{n_s}} = 1$ after the first solving phase, we add constraint $a_i^{X_{n_s}} = 1$ to the model and run the optimizing phase again, until the model is infeasible. Then we remove the constraint $a_i^{X_{n_s}} = 1$. The results of each optimizing phase will be output.

## 4 Applications

In this section, we show some applications of our automatic search model. We applied our algorithm to SKINNY, CRAFT, QARMA, PRINCE, PRINCEv2, and MIDORI block ciphers. All the results are shown in Table 1.

### 4.1 Differential fault analysis on SKINNY-n-n and SKINNY-n-2n

#### 4.1.1 DFA on SKINNY-n-n

By utilizing our automatic search model, we are able to develop better DFAs on SKINNY-n-n, with fault injections at the beginning of round $R - 8$. Denote the $i$-th cell in the master tweakey state as $tk_i$. In this section, we show the attack for SKINNY-128-128 by steps as an example.

- **Injection 1:** As shown in Figure 5, firstly we inject random byte faults at $X_{R-8}[13]$, and query for the correct and faulty ciphertext pairs. According to the matching rules we build for SKINNY, we get two differential equations that correspond to filter phase Filter 1 and Filter 2, i.e. Filter 1 uses the differential equation $\Delta X_{R-2}[1] = \Delta X_{R-2}[13]$ to filter the wrong key bytes and Filter 2 uses $\Delta X_{R-2}[4] = \Delta X_{R-2}[12]$ to do that.

**Filter 1:** We consider the filter process of

$$\Delta X_{R-2}[1] = \Delta X_{R-2}[13]. \quad (4)$$

Use $\tilde{X}_{R-2}$ to denote the faulty state of $X_{R-2}$ with fault injections. The differential state $\Delta X_{R-2}[1] = X_{R-2}[1] \oplus \tilde{X}_{R-2}[1]$ can be calculated with the correct and faulty ciphertext pairs by guessing the tweakey cells $\{tk_2, tk_9, tk_{13}\}$, and the differential state $\Delta X_{R-2}[13]$ can be calculated by guessing $\{tk_3, tk_{10}, tk_{15}\}$. Equ. (4) provides a filter of $2^{-8}$. Therefore, we require 6 pairs of correct and faulty ciphertexts with differences at $X_{R-8}[13]$ to recover the tweakey bytes $\{tk_2, tk_3, tk_9, tk_{10}, tk_{13}, tk_{15}\}$.

Such that we choose 6 plaintexts, and inject random byte faults at $X_{R-8}[13]$ to get the corresponding ciphertexts, and guess the tweakey bytes to compute the differential state $\Delta X_{R-2}[1] \oplus \Delta X_{R-2}[13]$. After the filter of Equ. (4), there is about 1 possible value of $\{tk_2, tk_3, tk_9, tk_{10}, tk_{13}, tk_{15}\}$ remaining. We use hash tables to reduce the complexity of this process. Specifically, we calculate the differential state $\Delta X_{R-2}[1]$ for 6 pairs of correct and faulty ciphertexts, and store the corresponding tweakey bytes $\{tk_2, tk_9, tk_{13}\}$ indexed by the value $\Delta X_{R-2}[1](1 \leq j \leq 6)$ of the 6 pairs. Then we calculate $\Delta X_{R-2}[13]$
Injection 2 (Filter 4)

Subkey cells need not be guessed

Determined by Set1 ∩ Set2

Figure 5: DFA on SKINNY-128-128.
for all 6 pairs of ciphertexts with all possible hypotheses of \{tk_3, tk_{10}, tk_{15}\}. If there is an intersection with the index, the 6-byte key is potentially correct. We expect to retrieve one correct value of the 6 tweakey bytes \{tk_2, tk_3, tk_9, tk_{10}, tk_{13}, tk_{15}\} due to the 6 pairs match of \(\Delta X_{R-2}[1] = \Delta X_{R-2}[13]\). The time complexity of this step is about \(2^{24}\) and the memory cost is also about \(2^{24}\).

Filter 2: We use the differential equation \(\Delta X_{R-2}[4] = \Delta X_{R-2}[12]\) to filter the tweakey bytes, similar to the process in Filter 1. With already recovered \{tk_2, tk_9, tk_{13}\} in the previous phase, we guess \(tk_{12}\) to compute the value of differential state \(\Delta X_{R-2}[4]\), and guess \{tk_4, tk_{11}, tk_{12}\} to compute the value of differential state \(\Delta X_{R-2}[12]\). We build a hash table indexed by \(\Delta X_{R-2}[4]\) for the 6 ciphertext pairs obtained in Filter 1 to filter the key bytes, where about 1 candidate remaining. The time complexity of this step is about \(2^{24}\) and the memory cost is about \(2^8\).

- **Injection 2**: In this stage, we inject random byte faults at \(X_{R-8}[14]\) and filter the key bytes. This is a two-step process involving Filter 3 and Filter 4.

Filter 3: We treat the differential equation \(\Delta X_{R-2}[2] = \Delta X_{R-2}[14]\) by guessing \{tk_0, tk_{12}, tk_{15}\} to compute \(\Delta X_{R-2}[2]\), and guessing \{tk_5, tk_8, tk_{13}\} to compute \(\Delta X_{R-2}[14]\). Note that we have retrieved \{tk_{12}, tk_{13}, tk_{15}\} in Injection 1. Such that we can build hash tables to filter \{tk_0, tk_5, tk_8\} with 3 pairs of correct and faulty ciphertexts with fault injections at \(X_{R-8}[14]\). This step costs about \(2^{16}\) time complexity and \(2^8\) memory.

Filter 4: We compute \(\Delta X_{R-2}[13]\) with known key bytes \{tk_3, tk_{10}, tk_{15}\}, and compute \(\Delta X_{R-2}[5]\) using known key bytes \{tk_0, tk_{10}, tk_{12}, tk_{15}\} and guessed \(tk_{14}\). Then we can filter the value of \(tk_{14}\) by \(\Delta X_{R-2}[13] = \Delta X_{R-2}[5]\) using the 3 ciphertext pairs in Filter 3. This costs \(2^8\) time complexity. So far, we have retrieved the value of \{tk_0, tk_2, tk_3, tk_4, tk_5, tk_8, tk_9, tk_{10}, tk_{11}, tk_{12}, tk_{13}, tk_{14}, tk_{15}\}.

- **Exhaustively Search**: For the remaining 3 bytes of tweakey that we have not guessed, we test all possible values and check the plaintext and ciphertext. This step costs a time complexity of about \(2^{24}\).

The overall time and memory complexities of our attack are about \(2^{24}\), and we need to inject 9 random bytes faults to recover the full key.

For SKINNY-64-64, the process is similar, which only alters the positions of tweakey cells guessed because of the difference in encryption round. And we can inject faults at the beginning of round \(R - 9\) to get a more powerful differential fault analysis.

Similar to the attack on SKINNY-128-128, we inject random faults at \(X_{R-9}[13]\) and get the differential equation \(\Delta X_{R-3}[1] = \Delta X_{R-3}[13]\). Then we guess the tweakey nibbles in \(\text{Set}_1\) and \(\text{Set}_2\) to get the values of the left and the right side of the equation respectively, where \(\text{Set}_1 = \{tk_2, tk_5, tk_9, tk_{11}, tk_{12}, tk_{15}\}\), \(\text{Set}_2 = \{tk_6, tk_7, tk_{10}, tk_{13}, tk_{15}\}\). We can utilize 10 pairs of correct and faulty ciphertexts with differences in \(X_{R-9}[13]\) to reduce the number of possible values of the 10 tweakey nibbles. We choose 10 random plaintexts and inject random faults at \(X_{R-9}[13]\). This phase costs about \(2^{23}\) time and \(2^{20}\) memory. And for the remaining 6 nibbles \{tk_0, tk_1, tk_3, tk_4, tk_8, tk_{14}\}, we exhaustively search and test with the correct plaintext and ciphertext, which costs \(2^{24}\) time for encryption.

4.1.2 Simulation results

We perform the simulation experiments according to our methods in Sect. 4.1.1. We randomly choose keys and repeat the attack process for 1000 times. In each experiment, we vary the number of injected faults in each filter step and calculate the average number of key candidates after filtering. Our results are summarized in Figure 6, where the x-axis
represents the number of fault injections in each filter step, and the \( y \)-axis represents the average number of remaining key candidates across 1000 simulations. For example, the red triangle point with the \( x \)-axis coordinate of 3 represents the number of remaining keys after filtering by 3 pairs of ciphertexts in Filter 2, which is \( 2^{59.2} \). In our experiments, with all 9 fault injections, the number of remaining key candidates after Filter 1-4 is always \( 2^{24} \), which validates that our attack is effective. All simulations are performed on a PC using C code, and each key recovery operation takes a few minutes. Our codes are available at https://github.com/yqy-yu/MITM-DFA.

<table>
<thead>
<tr>
<th>Injection 1</th>
<th>Injection 2</th>
<th>Injection 3</th>
<th>Injection 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{24} )</td>
<td>( 2^{32} )</td>
<td>( 2^{56} )</td>
<td>( 2^{80} )</td>
</tr>
</tbody>
</table>

Figure 6: Simulation results

4.1.3 DFA on SKINNY-n-2n

The attack in Sect. 4.1.1 targets at the SKINNY-n-n, where only \( TK_1 \) is used. SKINNY also supports versions with \( t = 2n \) and \( t = 3n \). In the TWEAKEY framework, users can choose what part of the tweakkey serves as input key material or tweak material, and \( TK_1 \) is recommended for processing the public tweak material. In this section, we consider the case of SKINNY-n-2n, where \( TK_1 \) is used for tweak and \( TK_2 \) is used for key material.

According to our injection model in Figure 5, we can extract \( \mathcal{L}_1^{R-3}(TK_1) \oplus \mathcal{L}_2^{R-3}(TK_2)[1] \), \( \mathcal{L}_1^{R-2}(TK_1) \oplus \mathcal{L}_2^{R-2}(TK_2)[0,6] \) and \( \mathcal{L}_1^{R-1}(TK_1) \oplus \mathcal{L}_2^{R-1}(TK_2)[4,5,7] \) in Filter 1, where \( \mathcal{L}_1 \) and \( \mathcal{L}_2 \) denote the linear operations on \( TK_1 \) and \( TK_2 \). \( \mathcal{L}_1 \) only represents the permutation \( P_T \) whereas \( \mathcal{L}_2 \) denotes the combination of \( P_T \) and LFSR on \( TK_2 \). Since the tweak \( TK_1 \) can be seen as public, we can compute \( \mathcal{L}_2^{R-3}(TK_2)[1] \), \( \mathcal{L}_2^{R-2}(TK_2)[0,6] \) and \( \mathcal{L}_2^{R-1}(TK_2)[4,5,7] \) straightforwardly from Filter 1. Then we can determine the corresponding bytes in \( TK_2 \) by inverting the LFSR and the permutation \( P_T \) round by round. The same for other filter phases.

In the actual situation, the security does not rely on the privacy of tweak material, and it is also unrealistic to keep privacy in practice. So we can retrieve the key material for SKINNY-n-2n as same as for SKINNY-n-n. It indicates that the tweak material does not provide any extra protection against fault attack if keeping fixed.

4.2 Differential fault attack on CRAFT

4.2.1 Description of CRAFT

CRAFT [BLMR19] is a lightweight tweakable block cipher with a 64-bit block size, a 128-bit key \( K \), and a 64-bit tweak \( T \). The cipher’s internal state can be represented as a \( 4 \times 4 \) square array of nibbles or as a 16-nibble vector by concatenating the rows of the square array. Use the index \( 4 \times i + j \) to denote the nibble at row \( i \) and column \( j \) of the \( 4 \times 4 \) array, where \( 0 \leq i \leq 3 \), \( 0 \leq j \leq 3 \).

For CRAFT, each round function applies five involutory round operations: SubSbox(SB), MixColumn(MC), PermutedNibbles(PN), AddConstant(ARC) and AddTweakey(ATK). The
last round omits SubSbox (SB) and PermuteNibbles (PN). The r-th round function $R_r$ is defined as $R_r = SB \circ PN \circ ATK \circ ARC \circ MC (1 \leq r \leq 31)$, and the last round $R_{12} = ATK \circ ARC \circ MC$. The involutory binary matrix $M$ used in MixColumn (MC) is non-MDS, where

$$M = \begin{pmatrix}
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{pmatrix}.$$ 

In the key schedule, four 64-bit tweakes $TK_0, TK_1, TK_2, TK_3$ are derived from $K = (K_0 || K_1)$ and $T$, as $TK_0 = K_0 \oplus T$, $TK_1 = K_1 \oplus T$, $TK_2 = K_0 \oplus Q(T)$, $TK_3 = K_1 \oplus Q(T)$, where $Q$ is a permutation and $TK_{(i-1) \text{mod } 4}$ is applied in the $i$-th round.

When guessing the key cells, we can choose the nibbles in equivalent subkeys, i.e., we can adjust the operations in round function as $R_i = SB \circ PN \circ ARC \circ MC \circ ATK'$, the key used in $ATK'$ is the equivalent subkey $MC^{-1}(TK_i)$.

### 4.2.2 DFA on CRAFT

CRAFT [BLMR19] is designed to be efficiently protected in its implementations against differential fault analysis. However, we assume that the target implementation of CRAFT does not use such countermeasures, as studied in [RVR22]. In round $r$, denote the state at the beginning of round $r$ as $X_r$, the state after $MC$ as $Y_r$, and the internal state after $PN$ as $Z_r$. Further, $A[i]$ denotes the $i$-th nibble of the state $A$, where $0 \leq i \leq 15$.

- **Injection 1:** As shown in Figure 7, we inject random nibble faults at $Z_{R-10}[7]$, the differential equation $\Delta Y_{R-3}[5] = \Delta Y_{R-3}[13]$ can be derived. And we guess the subkey nibbles $\{MC^{-1}(K_0)[3, 4, 13], MC^{-1}(K_1)[0, 1, 2, 7, 10, 14]\}$ to compute $\Delta Y_{R-3}[13]$ by partial decryption. Similarly, guess $\{MC^{-1}(K_0)[5], MC^{-1}(K_1)[2, 9]\}$ to obtain $\Delta Y_{R-3}[5]$.

Because the differential equation provides a filter of one nibble, we can retrieve the subkey nibbles with 11 pairs of correct and faulty ciphertexts. We choose 11 plaintexts and obtain the ciphertexts, then inject random faults at $Z_{R-10}[7]$ to each and query 11 faulty ciphertexts. Filtering the subkey nibbles with differential equation $\Delta Y_{R-3}[5] = \Delta Y_{R-3}[13]$, there is about 1 value of the key nibbles remaining. We need to inject 11 random nibble faults. The time complexity is about $2^{30}$ and the memory cost is about $2^{12}$.

![Figure 7: DFA on CRAFT.](image-url)
Injection 2: In this step, we inject random nibble faults at $Z_{R-10}[5]$, and get the differential equation $\Delta Y_{R-3}[7] = \Delta Y_{R-3}[15]$. We can filter the subkey nibbles $\{MC^{-1}(K_0)[7], MC^{-1}(K_1)[11]\}$ and $\{MC^{-1}(K_0)[1, 6, 15], MC^{-1}(K_1)[3, 5, 8, 12]\}$ with 9 pairs of correct and faulty ciphertexts. The time complexity is about $2^{28}$ and the memory complexity is about $2^8$.

Injection 3: Then we inject random nibble faults at $Z_{R-10}[6]$, filter the subkey nibbles $\{MC^{-1}(K_0)[2, 12], MC^{-1}(K_1)[4, 13]\}$ by $\Delta Y_{R-3}[4] = \Delta Y_{R-3}[12]$ with 4 pairs of correct and faulty ciphertexts. The time cost is $2^{16}$.

Injection 4: Finally we inject random nibble faults at $Z_{R-10}[4]$, filter the subkey nibbles $\{MC^{-1}(K_0)[0, 14], MC^{-1}(K_1)[6, 15]\}$ by $\Delta Y_{R-3}[6] = \Delta Y_{R-3}[14]$ with 4 pairs of correct and faulty ciphertexts. And then exhaustively search the remaining four subkey nibbles, the time cost is $2^{16}$.

The overall time complexity of the attack is about $2^{36}$, and the memory cost is about $2^{12}$. The number of faults we need to inject in all phases is $11 + 9 + 4 + 4 = 28$.

4.3 Differential fault attack on QARMA

4.3.1 Description of QARMA

QARMA [Ava17] is a family of lightweight tweakable block cipher. It supports two kinds of block sizes $n = 64$ and $n = 128$, denoted by QARMA-64 and QARMA-128. Each version applies a master key of $2n$ bits. QARMA uses an Even-Mansour scheme with a keyed pseudo-reflector. The first $r$ rounds of the cipher use the forward round function $R(IS, tk)$, which is composed of four operations in order: AddRoundTweakey, ShuffleCells($\tau$), MixColumns($M$) and SubCells($S$). The $(r+1)$-th round function uses the backward round function $R^-(IS, tk)$, which is the inverse of $R(IS, tk)$. The round function used in the first and last rounds omits ShuffleCells and MixColumns. The matrix $M$ used in MixColumns is defined as

$$M_4 = \text{circ}(0, \rho, \rho^2, \rho) \text{ and } M_8 = \text{circ}(0, \rho, \rho^4, \rho^5),$$

where $M_4$ is used for QARMA-64 and $M_8$ is used for QARMA-128. $\rho$ can be seen as a simple circular left rotation of the bits. The $2n$-bit $K$ is partitioned as $\omega_0|k_0$. In encryption, $k_1 = k_0$ is used for each round and whitening keys $\omega_0$ and $\omega_1 = \omega(\omega_0)$ are added at the beginning and the end, respectively. In the key-recovery phase, we will guess the equivalent key $M(\tau(k_0))$.

4.3.2 DFA on QARMA-64

Because the matrix $M$ used in MixColumns is different in each version, QARMA-64 and QARMA-128 have different differential matching rules. In this section, we show how to achieve the key recovery attack on QARMA-64. Part of the attack is shown in Figure 8.

- **Injection 1**: Inject random nibble faults at $X_{R-4}[15]$ ($Z_{R-5}[13]$), filter the subkey nibbles $\{M(\tau(k_0))[10], k_0 \oplus \omega_1[6, 9, 12]\}$ and $\{M(\tau(k_0))[15], k_0 \oplus \omega_1[7, 8, 13]\}$ by the differential equation $\Delta Z_{R-2}[4] = \Delta Z_{R-2}[12]$ with 8 pairs of correct and faulty ciphertexts.

- **Injection 2**: Inject random nibble faults at $X_{R-4}[12]$ ($Z_{R-5}[6]$), filter the subkey nibbles $\{M(\tau(k_0))[5], k_0 \oplus \omega_1[4, 11, 14]\}$ and $\{M(\tau(k_0))[0], k_0 \oplus \omega_1[5, 10, 15]\}$ by $\Delta Z_{R-2}[8] = \Delta Z_{R-2}[0]$ with 8 pairs of correct and faulty ciphertexts.

- **Injection 3**: Inject random nibble faults at $X_{R-4}[0]$ ($Z_{R-5}[0]$), filter the subkey nibbles $\{M(\tau(k_0))[6], k_0 \oplus \omega_1[3]\}$ and $\{M(\tau(k_0))[3], k_0 \oplus \omega_1[18]\}$ by $\Delta Z_{R-2}[2] = \Delta Z_{R-2}[10]$ with 4 pairs of correct and faulty ciphertexts.
Injection 4: Inject random nibble faults at $X_{R-4}[6]$ ($Z_{R-5}[2]$), filter the subkey nibbles $\{M(\tau(k_0))[1], k_0 \oplus \omega_1[1]\}$ and $\{M(\tau(k_0))[4], k_0 \oplus \omega_1[0]\}$ by $\Delta Z_{R-2}[5] = \Delta Z_{R-2}[13]$ with 4 pairs of correct and faulty ciphertexts.

Injection 5 and Injection 6 respectively inject random faults at $X_{R-4}[3]$ and $X_{R-4}[4]$. Injection 5 filter $M(\tau(k_0))[9, 12]$ by $\Delta Z_{R-2}[6] = \Delta Z_{R-2}[14]$, and Injection 6 filter $M(\tau(k_0))[11, 14]$ by $\Delta Z_{R-2}[1] = \Delta Z_{R-2}[9]$. Each filter phase needs 2 pairs of correct and faulty ciphertexts.

After the six injection phases, we exhaustively search for the remaining 4 subkey nibbles $M(\tau(k_0))[2, 7, 8, 13]$ and retrieve subkey $k_0$ by linear transformation $\tau \circ M$. Then we compute $\omega_1$ by $k_0$ and $k_0 \oplus \omega_1$. The overall time complexity of our attack is about $2^{16}$ and the memory complexity is about $2^{32}$. It requires the injection of 28 nibble faults.

4.3.3 DFA on QARMA-128

In this section, we show the process of our fault attack on QARMA-128 with $2^{32}$ time complexity and $2^{32}$ memory complexity.

Injection 1: We inject random faults at $X_{R-3}[15]$ and get following equation system:

$$\begin{align*}
\Delta Z_{R-2}[5] &= \rho^3 \cdot \Delta Z_{R-2}[9], \\
\rho \cdot \Delta Z_{R-2}[0] &= \Delta Z_{R-2}[12], \\
\rho^3 \cdot \Delta Z_{R-2}[3] &= \Delta Z_{R-2}[15], \\
\rho^4 \cdot \Delta Z_{R-2}[1] &= \Delta Z_{R-2}[9], \\
\Delta Z_{R-2}[11] &= \rho \cdot \Delta Z_{R-2}[15], \\
\rho^4 \cdot \Delta Z_{R-2}[4] &= \Delta Z_{R-2}[12].
\end{align*}$$

Each equation provides a one-byte filter for involved subkey bytes. For example, we choose 8 plaintexts and inject random faults to get the correct and faulty ciphertexts. We can then guess the subkey bytes $\{M(\tau(k_0))[1], k_0 \oplus \omega_1[1, 4, 14]\}$ to compute $\Delta Z_{R-2}[5]$ and guess $\{M(\tau(k_0))[14], k_0 \oplus \omega_1[3, 6, 12]\}$ to compute $\Delta Z_{R-2}[14]$ for the 8 pairs. By filtering with $\rho^3 \cdot \Delta Z_{R-2}[5] = \Delta Z_{R-2}[9]$, we can retrieve the value of the 8 subkey bytes. Similarly, we can retrieve the subkey bytes $M(\tau(k_0))[0, 2, 8, 10, 11, 13, 15]$ and $k_0 \oplus \omega_1[0, 2, 5, 7, 8, 9, 10, 11, 13, 15]$ with the remaining equations. We can use the 8 pairs of correct and faulty ciphertexts obtained in the first step to accomplish this.

Injection 2: Then we inject random faults at $X_{R-3}[6]$ and get 2 differential equations for a further filter of subkey bytes,

$$\rho^3 \cdot \Delta Z_{R-2}[2] = \Delta Z_{R-2}[14], \quad \Delta Z_{R-2}[7] = \rho^4 \cdot \Delta Z_{R-2}[15].$$
Only at most one unknown subkey byte is involved in one hand of each equation. So we choose 2 plaintexts and inject faults to get the correct and faulty ciphertexts. We expect to get the right value of $M(\pi(k_0))[6, 7, 9]$. Then we exhaustively search for the remaining 4 subkey bytes $M(\pi(k_0))[3, 4, 5, 12]$ to retrieve the full key. The overall time complexity of this attack is about $2^{32}$ and the memory cost is $2^{32}$. The number of faults we need to inject in the attack is 10.

4.4 Differential fault attack on PRINCE and PRINCEv2

4.4.1 Description of PRINCE

PRINCE [BCG$^+$12] is a lightweight block cipher that follows FX-construction with a 64-bit block size and a 128-bit key size. The 128-bit key $K = K_0||K_1$ is split into two 64-bit parts. $K_0$ is used as a whitening key and $K_1$ is used as round keys for the core of the structure, named as PRINCEcore. The round function $R$ of PRINCEcore applies an S-box layer $SB$, followed by a linear layer consisting of a MixColumns operation $MC$ and a ShiftRows $SR$. $SR$ is the same as AES, and the matrices in $MC$-layer are built from the following four matrices:

$$m_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, m_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, m_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, m_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

In $MC$-layer, bit-wise matrix $M_0$ is multiplied with the first and the last columns of the internal state, and $M_1$ is multiplied with the second and the third columns.

$$M_0 = \begin{pmatrix} m_1 & m_2 & m_3 & m_4 \\ m_2 & m_3 & m_4 & m_1 \\ m_3 & m_4 & m_1 & m_2 \\ m_4 & m_1 & m_2 & m_3 \end{pmatrix}, M_1 = \begin{pmatrix} m_2 & m_3 & m_4 & m_1 \\ m_3 & m_4 & m_1 & m_2 \\ m_4 & m_1 & m_2 & m_3 \\ m_1 & m_2 & m_3 & m_4 \end{pmatrix}.$$
The last five rounds apply the inverse of the forward round function $R^{-1}$, and the middle layer applies $R'$. The round function $R, R'$ and $R^{-1}$ are shown as:

$$R = SR \circ MC \circ SB, \quad R' = SB^{-1} \circ MC \circ SB, \quad \text{and} \quad R^{-1} = SB^{-1} \circ MC \circ SR^{-1}.$$  

PRINCEv2 [BEK+20] modifies the middle layer and key schedule of PRINCE. The 128-bit key is departed into $K = (K_0||K_1)$ and alternately used in each function. The subkeys used in the last two rounds are $K_0 \oplus \alpha$ and $K_1 \oplus \beta$, where $\alpha$ and $\beta$ are constants. The process for retrieving the full key is similar to that for PRINCE.

### 4.4.2 DFA on PRINCE

We can recover the full key of PRINCE with random faults injected at $Z_{R-3}[0]$. As shown in Figure 10, we can build the following equation system according to differential characters:

$$\begin{align*}
m_1 \cdot \Delta Z_{R-1}[3] &= m_4 \cdot \Delta Z_{R-1}[7], & m_2 \cdot \Delta Z_{R-1}[3] &= m_4 \cdot \Delta Z_{R-1}[11], \\
m_3 \cdot \Delta Z_{R-1}[3] &= m_4 \cdot \Delta Z_{R-1}[15], & m_2 \cdot \Delta Z_{R-1}[7] &= m_1 \cdot \Delta Z_{R-1}[11], \\
m_3 \cdot \Delta Z_{R-1}[7] &= m_1 \cdot \Delta Z_{R-1}[15], & m_3 \cdot \Delta Z_{R-1}[11] &= m_2 \cdot \Delta Z_{R-1}[15].
\end{align*}$$  

(7)

We guess $\{K_1[11], K_0' \oplus K_1[1, 5, 9, 13]\}$ to compute $\Delta Z_{R-1}[11]$ and filter the values of $\{K_1[15], K_0' \oplus K_1[2, 6, 10, 14]\}$ by $m_3 \cdot \Delta Z_{R-1}[11] = m_2 \cdot \Delta Z_{R-1}[15]$. This equation provides a 2-bit filter for each correct and faulty ciphertext pair. With 10 pairs of correct and faulty ciphertext pairs, we get 20-bit information on these 10 subkey nibbles.

Similarly, we can get 20-bit information of $\{K_1[3, 7], K_0' \oplus K_1[0, 3, 4, 7, 8, 11, 12, 15]\}$ by the filter of $m_4 \cdot \Delta Z_{R-1}[3] = m_4 \cdot \Delta Z_{R-1}[7]$. We can obtain further filters by other equations in Equ. (7). For example, $m_4 \cdot \Delta Z_{R-1}[7] = m_1 \cdot \Delta Z_{R-1}[15]$ provides one more bit filter for $\Delta Z_{R-1}[15]$ and $\Delta Z_{R-1}[7]$.

Then we can build equation systems for other columns of $\Delta Z_{R-1}$, like

$$\begin{align*}
m_4 \cdot \Delta Z_{R-1}[8] &= m_3 \cdot \Delta Z_{R-1}[12], & m_2 \cdot \Delta Z_{R-1}[0] &= m_1 \cdot \Delta Z_{R-1}[4]
\end{align*}$$  

(8)

to filter the subkey nibbles. We expect to get the correct value of $K_0' \oplus K_1$ and at least 48-bit information of $K_1$. We exhaustively search for the remaining possible values of $K_1$, which costs about $2^{10}$ time, and compute $K_0'$ to retrieve the full key. The overall time and memory complexities are both about $2^{20}$, with 10 fault injections.
4.5 Differential fault attack on MIDORI

4.5.1 Description of MIDORI

MIDORI [BBI+15] is a family of low-energy block cipher, which includes two versions with block size $n = 64$ and $n = 128$, i.e., MIDORI-64 and MIDORI-128. Both versions accept a 128-bit key $K$. For MIDORI-64, $K$ is denoted as two 64-bit parts $K = K_0 || K_1$. The whitening key is $W = K_0 \oplus K_1$ and the round key is $RK_r = K(r-1) \mod 2 \oplus \alpha_r$ for $1 \leq r \leq 15$. For MIDORI-128, $W = K$ and $RK_r = K \oplus \beta_r$ for $1 \leq r \leq 19$, where $\alpha_r$ and $\beta_r$ are constants. The round function of MIDORI consists of SubSbox(SB), ShuffleCell(SC), MixColumn(MC), and KeyAdd(AK). The ShuffleCell(SC) applies a cellwise permutation $P$ to the internal state, where $P = [0, 11, 6, 13, 10, 1, 12, 7, 5, 14, 3, 8, 15, 4, 9, 2]$. In MixColumn(MC) operation, each column of the state array is multiplied by the matrix $M$ cellwise, where $M = \text{circ}(0, 1, 1, 1)$. The last round only applies SubSbox(SB) operation and adds the whitening key $W = K$ to the internal state.

4.5.2 DFA on MIDORI

DFA on MIDORI-64. We firstly inject random faults at $X_{R-4}[2]$ as shown in Figure 11, and get three differential equations

$$\Delta X_{R-1}[5] = \Delta X_{R-1}[13], \Delta X_{R-1}[0] = \Delta X_{R-1}[4], \Delta X_{R-1}[7] = \Delta X_{R-1}[11]. \quad (9)$$

Each equation represents a filter step (Filter 1-3). Filter 1 guess $\{MC^{-1}(K_0)[8], K_0 \oplus K_1[0, 4, 12]\}$ to compute $\Delta X_{R-1}[5]$, and guess $\{MC^{-1}(K_0)[3], K_0 \oplus K_1[7, 11, 15]\}$ to compute $\Delta X_{R-1}[13]$, then filter the subkey nibbles by $\Delta X_{R-1}[5] = \Delta X_{R-1}[13]$. We expect to get one candidate of $\{MC^{-1}(K_0)[0, 1, 3, 7, 8, 13], K_0 \oplus K_1[0, 1, 3, 4, 5, 7, 8, 9, 11, 12, 13, 15]\}$ after Filter 1-3 with 8 pairs of correct and faulty ciphertexts.

Secondly, we inject faults at $X_{R-4}[9]$ and filter the subkey nibbles with equations

$$\Delta X_{R-1}[3] = \Delta X_{R-1}[15], \Delta X_{R-1}[8] = \Delta X_{R-1}[12]. \quad (10)$$

We expect to get one right value of $\{MC^{-1}(K_0)[6, 10, 11, 12], K_0 \oplus K_1[2, 6, 10, 14]\}$ with 5 pairs of correct and faulty ciphertexts (Filter 4-5).

Then we inject faults at $X_{R-4}[4]$ and filter $MC^{-1}(K_0)[2, 5, 9, 14]$ with the equations $\Delta X_{R-1}[1] = \Delta X_{R-1}[9]$ and $\Delta X_{R-1}[6] = \Delta X_{R-1}[14]$. We expect to get the right value with 2 pairs of correct and faulty ciphertexts.

Finally, we exhaustively search for the values of $MC^{-1}(K_0)[4, 15]$ and compute $K_0$ and $K_1$ to recover the full key. The overall time complexity is about $2^{16}$ and the memory complexity is $2^{16}$, and the number of faults we need to inject is 15.

DFA on MIDORI-128. For MIDORI-128, the key used in the last two rounds is the same except for the constant addition. We can recover $K[0, 1, 3, 4, 5, 7, 8, 9, 11, 12, 13, 15]$ according to the filter of Equ. (9) with faults injected to $X_{R-4}[2]$. To compute the value of each side in Equ. (9), we need to guess 3 bytes of $K$, for example, $K[0, 1, 12]$ for $\Delta X_{R-1}[5]$. This phase can be done with 6 pairs of correct and faulty ciphertexts. Then, we inject faults at $X_{R-4}[9]$ and filter the subkey bytes according to Equ. (10). We expect to get the right value of the full key with a filter of 3 pairs. Thus, the overall time complexity of the attack on MIDORI-128 is about $2^{24}$, and the memory cost is also $2^{24}$, with 9 fault injections.

4.6 Extended Attack on AES-192

In [DFL11], Derbez et al. extended the DFA attack on AES-128 to AES-192. The DFA attack on AES-128 is stated in Sect. 2.1. The extended DFA attack on AES-192 is briefly given as follows, Figure 1 represents the last four rounds of AES-192.
Step I: They first recover the key cells of $K_{12}$ for AES-192 using the same MitM algorithm as AES-128. Substitute $R$ with 12, and denote differential state $X_{11} \oplus \hat{X}_{11}$ as $\Delta X_{11}$. Filter the values of $\{K_{12}[1, 4, 11, 14, U_{11}[7]], \{K_{12}[2, 5, 8, 15, U_{11}[10]\}, \{K_{12}[0, 7, 10, 13, U_{11}[0]\}$ and $\{K_{12}[3, 6, 9, 12, U_{11}[13]\}$ according to the equations between $\Delta X_{11}[0], \Delta X_{11}[1], \Delta X_{11}[2]$ and $\Delta X_{11}[3]$. The time of this step is the same as the DFA on AES-128, which is $3 \times 2^{40}$.

Step II: After recovering the 128-bit subkey $K_{12}$, in order to recover the full 192-bit key, they peel off the last round of AES-192 to perform a 3-round (Round 9-11) differential fault attacks following the idea of Piret and Quisquater [PQ03] with the same correct and faulty pairs. Therefore, an additional time complexity is required to implement Piret and Quisquater’s attack, which is estimated as $2^{40}$ “Piret and Quisquater resolution” by Derbez et al. [DFL11].

4.6.1 Our Improved Step II

We introduce a new MitM DFA on AES-192, which uses the same Step I as Derbez et al. [DFL11] to recover $K_{12}$. But we do not use Piret and Quisquater’s attack (Step II) to recover the remaining key bits.

A new Step II: After recovering $K_{12}$, we also peel off the last round. In order to recover the full 192-bit key, we need to recover the last two columns of $K_{11}$, which is equivalent to recovering $U_{11}[8-15]$. Among those bytes, $U_{11}[10, 13]$ are already recovered in Step I.

After peel off Round 12, we can derive $\Delta X_{11}[8] = 3 \cdot \Delta X_{11}[11]$, which is equivalent to

$$\Delta X_{11}[8] = 3 \cdot \Delta X_{11}[11]$$

where $A$ and $B$ are known values at this stage and only depend on the correct ciphertext and $K_{12}$, similar to $\hat{A}$ and $\hat{B}$. Then, we use Eq. (11) as a filter to build a local MitM attack, where two correct-faulty ciphertext pairs will provide a filter of $2^{-16}$. Therefore, only one candidate of $U_{11}[8, 15]$ will remain. The time complexity of local MitM is $2 \times 2^8$ with about $2^8$ memory.

Similarly, we can recovery $U_{11}[11, 14]$ and $U_{11}[9, 12]$ with matching equations like Eq. (11). The last two columns of $K_{11}$ are recovered. The total time complexity of our new Step II is $6 \times 2^8 = 2^{10.6}$ with about $2^8$ memory, which is smaller than Derbez et al.’s
Step II. Two correct-faulty ciphertext pairs needed in our new Step II can reuse the pairs from Step I, and no additional fault injections are needed here. However, since Step I dominates the overall time complexity of the DFA on AES-192, the overall time complexity of the full attack is the same as Derbez et al.’s [DFL11]. This process can be also applied to AES-256 to improve Step II.

5 Discussion

In order to achieve lightweight, many lightweight ciphers adopt sparser diffusion layers. For example, MIDORI, PRINCE and QARMA adopt the so-called 4 × 4 almost MDS layer. Note that the branch numbers (the smallest nonzero sum of active inputs and outputs of the matrix) of MDS and almost MDS matrices are 5 and 4. Therefore, the algorithms need more rounds to achieve full diffusion (any input bit nonlinearly affects all the state bits). For example, MIDORI needs 3 rounds to achieve full diffusion while AES needs 2 rounds. Ciphers with more lightweight diffusion layers, like SKINNY and CRAFT, respectively need 6 and 7 rounds to achieve full diffusion.

The main idea behind MitM DFA is to explore the potentially low diffusion of the target ciphers. The lower diffusion layers can cause more rounds that need to be protected against DFAs. With the MitM approach, a differential matching rule (e.g., Eq. (2)) exists with higher probability after more rounds of diffusion for one-byte faults, which will act as the matching to filter the wrong key guessing. Besides, with the lower diffusion layers, the number of key bits involved in the two neutral key sets (e.g., Set1 and Set2 in Figure 2) are smaller. It dominates the time complexity of the MitM DFAs, because the keys in Set1 (and Set2) must be enumerated to compute and filter with the differential equation (e.g., Eq. (2)). These are the reasons behind the attacks summarized in Table 1, where for SKINNY and CRAFT, the MitM DFA can achieve 9 and 10 rounds, but for MIDORI, PRINCE and QARMA, the attack can only work on 4 or 5 rounds.

Countermeasures The double-check mechanism is a common countermeasure against fault injection attacks [MSY06, ML08, JMR07, BBK+10]. The crucial operations of encryption devices, which are vulnerable to fault analysis, should run twice. If the results of the two executions match each other, the results are credible. Indeed, this mechanism is always accompanied by a loss of efficiency. Our improved DFAs in Sect. 4 provide new insight into how many rounds of encryption devices need to be protected. For example, we suggest protecting at least the last 9 rounds of SKINNY-128-128, with the double-check mechanism, the running procedure of SKINNY-128-128 is shown below:

1. Compute State1 = Round1−31(Message, Key), State2 = State1.
2. Compute Res1 = Round32−40(State1, Key), and Res2 = Round32−40(State2, Key).
3. After a random delay, check whether Res1 = Res2. If yes, output Res1 as ciphertext, else discard the result.

The number of rounds for SKINNY-128-128 is 40, in our fault detection scheme, 49 rounds need to be implemented. Such that the scheme requires about 49/40 computational sources of SKINNY-128-128.

6 Conclusion

In this paper, we present the MILP-based automatic tools for MitM DFAs and apply to SKINNY, CRAFT, QARMA, PRINCE, MIDORI and AES-192/256. We make full use of the differential matching rules and reach better key-recovery attacks for these block ciphers in
practical time. We achieve fault attacks with faults injected in earlier rounds, which imply that more rounds of the encryption devices of these ciphers should be protected against DFA.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their valuable comments. This work is supported by the National Key R&D Program of China (2022YFB2702804, 2018YFA0704701), the Natural Science Foundation of China (62272257, 62072270), Shandong Key Research and Development Program (2020ZLYS09), the Major Scientific and Technological Innovation Project of Shandong, China (2019JZZY010133), the Major Program of Guangdong Basic and Applied Research (2019B030302008), Key Research Project of Zhejiang Province, China (2023C01025).

References


[DHS+21] Xiaoyang Dong, Jialiang Hua, Siwei Sun, Zheng Li, Xiaoyun Wang, and Lei Hu. Meet-in-the-middle attacks revisited: Key-recovery, collision, and preimage attacks. In Tal Malkin and Chris Peikert, editors, Advances in Cryptology -


Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. In Chuankun Wu,
Qingyuan Yu et al.


