Automated Tool for Meet-in-the-Middle Attacks with Very Low Data and Memory Complexity (Full Version)

Mohammad Vaziri

The University of Edinburgh, Edinburgh, UK
mohammad.vaziri@ed.ac.uk,

Abstract. In this paper, we present a simple meet-in-the-middle attack that requires low data and memory resources. To evaluate the complexity of the attack, we also propose an automated tool that calculates the time, data, and memory complexities based on the suggested matching points. Our method operates at the bit level and employs a known-plaintext attack, with no constraints on the attacker's choice of data. We apply our tool on various lightweight block ciphers, including CRAFT, Midori, WARP, PRESENT, and ARADI. For CRAFT, our tool successfully identified an attack targeting 15 rounds using 3 known plaintexts. In the case of Midori64 and Midori128, the tool proposed attacks on 5 rounds with 16 known plaintexts and 7 rounds with 3 known plaintexts, respectively. For WARP, the tool discovered an attack on 18 rounds utilizing 7 known plaintexts. Additionally, for PRESENT80, the tool identified an attack on 6 rounds with 18 known plaintexts, and for ARADI, an attack on 5 rounds with 28 known plaintexts was determined.
Keywords: MITM Attacks, Automated Tools, Lightweight Cryptography, Key-recovery Attacks

1 Introduction

In the field of symmetric key cryptanalysis, meet-in-the-middle (MITM) attacks have drawn significant attention. Since the introduction of MITM by Diffie and Hellman [DH77], numerous MITM techniques have been developed to analyze the security of various cryptographic primitives, such as block ciphers, stream ciphers, and hash functions. MITM attacks involve identifying internal states in the middle of the primitive as matching points. The values at these matching points are computed independently in the forward and backward directions. By doing so, the attacker can retain partial key candidates that satisfy the matching conditions while discarding all others.

MITM attacks have been effectively applied to numerous designs using increasingly sophisticated techniques. Some of these techniques include Sieve-in-the-Middle [BC13], splice-and-cut [GLRW10], guess-and-determine [DSP07], bicliques [KRS12], 3-subset MITM [BR10], differential MITM [BDD$^+$23] and Demirci-Selçuk [DS08].

MITM techniques have also been automated by some researchers. In [DF16], the Demirci-Selçuk MITM attack was automated using general constraint programming. In [SS23], a new MITM technique was proposed using Mixed Integer Linear Programming (MILP), which can be applied to lightweight Substitution-Permutation-Network

(SPN) ciphers with a simple key schedule. Additionally, in [Sas18], Sasaki used MILP to optimize the 3-subset MITM attack on GIFT-64.

Since the US National Institute of Standards and Technology (NIST) announced a call for a new lightweight cryptographic standard in 2018 [oSN18], many lightweight cryptographic ciphers have been proposed. In lightweight ciphers, to make them suitable for lightweight platforms, the key schedule is often designed with slow diffusion or none at all. This design choice makes MITM attacks more feasible on such structures.

Our contributions. In this paper, we present a simple automated MITM attack which is easy to follow and implement. This technique is based on the relationship of the key bits at the matching points. The matching points depend only on the key bits that are nonlinearly involved, while the linearly involved key bits are canceled out by performing the differential in each direction. We introduce a tool [1] that models every bit and determines the relationship of key bits to each other.

Our proposed meet-in-the-middle attack introduces a notable balance between simplicity and efficiency, setting it apart from other complex techniques. While it may not achieve the highest number of rounds for key recovery, the simplicity of our approach results in exceptionally low data and memory complexity, making it highly accessible and easy to implement. Unlike many existing methods, our technique operates under a known-plaintext setting, eliminating the need for a chosen-plaintext attack and thus broadening its practical applicability. Despite its straightforward nature, which might limit its capability, the advantages in terms of data efficiency and ease of deployment position our technique as a valuable tool in cryptographic analysis.

We apply our tool to the block ciphers CRAFT [BLMR19], Midori [BBI+15], WARP [BBI+20], PRESENT [BKL+07], and ARADI [GMW24]. The complexities obtained using our tool are summarized in Table 1.

Table 1: Attacks overview. KP stands for Known Plaintext attack, and CP stands for Chosen Plaintext attack.

Target	Rounds	Data	Time(log_2)	Memory(log_2)	Attack Type	Reference
CRAFT	15/31	**3**	126.5/128	4	**KP**	Section 4.1
	20/31	2^{56}	126.9/128	109	CP	[MLC23]
Midori64	5/16	**16**	77/128	48	**KP**	Section 4.2
	12/16	$2^{55.5}$	125.5/128	106	CP	[LW17]
Midori128	7/20	**3**	126.3/128	0	**KP**	Section 4.2
WARP	18/41	**7**	125/128	6	**KP**	Section 4.3
	19/41	16	124.48/128	4	CP	[ZLW+23]
PRESENT80	6/31	**18**	68.2/80	15	**KP**	Section 4.4
	9/31	2^{12}	77/80	16	CP	[SS23]
ARADI	5/16	**28**	234.8/256	19	**KP**	Section 4.5

[1] The tool has been fully implemented, and all attacks have been empirically validated. The tool is available at: https://github.com/mohammadvaziri/Automated-Meet-in-the-Middle-Attacks-Tool.

Outline. The paper is organized as follows. In Section 2, we provide details about our attack methodology, outline our MITM algorithm, and explain how the time data and memory complexities are computed. Next, in Section 3, we describe how the operations in the round functions are modeled and how the nonlinearly and linearly dependent key bits are calculated for each bit. Additionally, we present an algorithm that summarizes the operation of our tool, where the output includes the suggested matching points as well as the time, memory, and data complexities. In Section 4, we apply our technique to perform key recovery attacks on CRAFT, Midori, WARP, PRESENT80 and ARADI. We conclude the paper in Section 5.

1.1 Notations

Throughout the paper, we will use the following notations:

$\mathcal{K}$	The set of all key bits.
K_f	The set of key bits involved in the forward direction at the matching points.
K_b	The set of key bits involved in the backward direction at the matching points.
K_u	The union of the set of all key bits in K_f and K_b ($K_u = K_f \cup K_b$).
K_c	The set of common key bits between K_f and K_b ($K_c = K_f \cap K_b$).
K_r	The set of key bits that are not included in K_u ($K_r = \mathcal{K} - K_u$).
$\mathcal{P}(\mathcal{K})$	The power set of the set $\mathcal{K}$.
$+$	The bitwise `XOR` operation.
$\|\|$	concatenation.

2 Background

In this section, we present a comprehensive overview of our MITM attack, along with the related formulas for time, data and memory complexities. In our approach, the meeting occurs at the bit level. Each bit within the internal rounds of the cipher algorithm retains crucial data, including the sets of key bits that are linearly and nonlinearly involved in the bit. Hereafter, the term *bit address* refers to the position of such encoded information. For the i-th bit of the internal state in round r, denoted as $x^r[i]$, we define the corresponding bit address as $X^r[i]$. In our methodology, essential information within each bit address is extracted using our automated tool in both forward and backward directions. The process of identifying the sets of linearly and nonlinearly dependent key bits at each bit address will be discussed in Section 3.

2.1 Description of Our MITM Approach

Our attack methodology is based on a known plaintext attack. To prepare the data for the attack, a set of pairs $D = \{(P_0, C_0), ..., (P_{d-1}, C_{d-1})\}$ of plaintext/ciphertext are randomly selected. The initial pair is designated as the primary pair. For each direction, the difference between the primary pair and the other pairs is computed at the matching points. These difference values are then used to check for a match. The purpose of calculating the differences at the matching points is to cancel the effect of linearly dependent key bits in both the forward and backward directions. The matching

points may consist of a single bit or multiple bits. The selection of a bit address as a matching point depends on several factors, which will be discussed in Section 3.

As discussed, each bit depends on specific key bits in both the forward and backward directions. It is essential that the bit addresses chosen as matching points depend on the same set of key bits in the forward direction and the same set of key bits in the backward direction.

To illustrate the rationale behind performing the difference at the matching points to eliminate the effect of linearly involved key bits, consider the cipher structure in Figure 1. Assume that the set of bit addresses $X=\{X[0],...,X[m-1]\}$ is chosen as the matching points. For a given pair of plaintext and ciphertext (P,C), and the key bits in the forward and backward directions, $k_f \in K_f$ and $k_b \in K_b$, we define the forward and backward path functions as $F_f(P,k_f)$ and $F_b(C,k_b)$, respectively. Assume that the sets of nonlinearly dependent key bits in K_f and K_b are denoted by K_f^n and K_b^n, and the sets of linearly dependent key bits are represented by K_f^l and K_b^l. In the following, we discuss why the difference between pairs at the matching points within the set $D=\{(P_0,C_0),...,(P_{d-1},C_{d-1})\}$ remains unaffected by any key bits in the sets K_f^l and K_b^l.

Assume that the Boolean function $F_f(P,k_f)$ in the forward direction at the matching points X is divided into two sub-functions, $G_n(P,k_f^n)$ and $G_l(k_f^l)$, such that $F_f(P,k_f)=G_n(P,k_f^n)+G_l(k_f^l)$, where $k_f^n \in K_f^n$ and $k_f^l \in K_f^l$. According to Relations 1, the differential between two plaintexts P_0 and P_1 at the matching points, cancels out the effect of the function $G_l(k_f^l)$. A similar conclusion can be drawn for the backward direction.

$$\begin{aligned} Xf_0 &= G_n(P_0,k_f^n)+G_l(k_f^l) \\ Xf_1 &= G_n(P_1,k_f^n)+G_l(k_f^l) \\ Xf_0+Xf_1 &= G_n(P_0,k_f^n)+G_n(P_1,k_f^n) \end{aligned} \tag{1}$$

It is important to note that the sub-function $G_l(k_f^l)$ can be derived independently of the plaintext vector P. Let $P=(p_0,...,p_{n-1})$ represent the plaintext bits and $K=(k_0,...,k_{n-1})$ the key bits introduced through a key schedule during the key addition phase. Since key addition is a linear operation, there exists no nonlinear interaction between bits, such as terms of the form $p_i k_j$ for $0 \le i,j < n$. Consequently, unless a nonlinear operation is applied to the bit addresses, the plaintext and key bits remain linearly dependent. In such cases, the sub-function $G_l(k_f^l)$ remains derivable. For a detailed discussion on the identification of linearly and nonlinearly dependent key bit sets, refer to Section 3.

Consequently, the matching point difference is just influenced by the nonlinearly dependent key bits involved in both the forward and backward directions. Hereafter, for simplicity, the sets K_f and K_b denote the sets of nonlinearly dependent key bits in the forward and backward directions, respectively, for the given bit address $X^r[i]$.

It is important to note that if we denote the differences of (Xf_0,Xf_1), (Xf_0,Xf_2), and (Xf_1,Xf_2) as ΔXf_{01}, ΔXf_{02}, and ΔXf_{12} respectively at the matching points, the difference ΔXf_{12} contains no useful information. This is because we have already

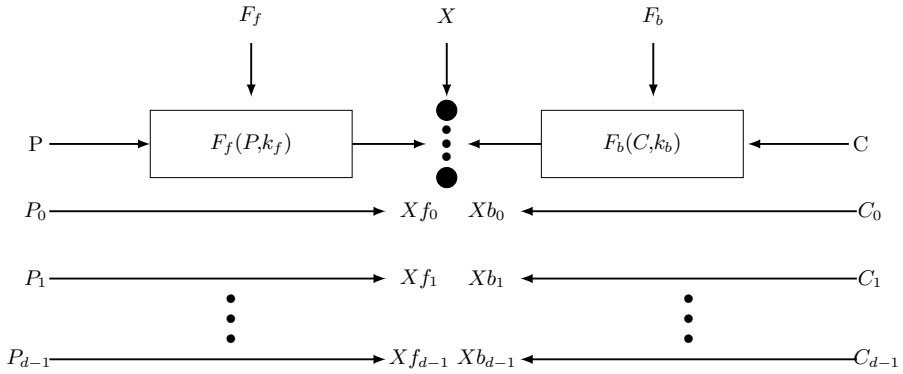

Fig. 1: General structure of the attack, where the forward and backward paths meet at the bit addresses $X=\{X[0],...,X[m-1]\}$. By taking the differences between the pairs $(Xf_0,Xf_1),...,(Xf_0,Xf_{d-1})$ and $(Xb_0,Xb_1),...,(Xb_0,Xb_{d-1})$, the effect of the linearly dependent key bits in both directions can be canceled.

utilized the differences ΔXf_{01} and ΔXf_{02} to filter certain candidates. Additionally, since the difference ΔXf_{12} can be derived from ΔXf_{01} and ΔXf_{02}, it cannot effectively filter key candidates. Hence, we designate one pair as the primary pair and compute the differences of other pairs in comparison to the primary one.

The attack is divided into two phases. In the first phase a table H is generated to store whole of the possible values of the differentials of the pairs in one of the directions and in the second phase, whole of the possible values of the other direction is computed and the found matches are evaluated. The role of the forward and backward directions can be interchanged. To minimize memory complexity, the direction with the smaller size of key bits involved should be selected for generating the H table.

It is notable that optimizing memory complexity can be achieved by considering the set of key bits K_c shared between the forward and backward directions. In this approach, for each $k_c \in K_c$, the computations for both phases can be performed independently, allowing the memory to be reset afterward.

In order to recover the remaining key bits K_r in our attack, during the second phase, once a match is found in the table H, a set of $T=\{(P'_0,C'_0),...,(P'_{t-1},C'_{t-1})\}$ data pairs, encrypted with the main master key, is required to verify the correctness of the guessed key bits. This means that each plaintext in T is encrypted using the guessed key bits $k_c\|k_f\|k_b\|k_r$, where $k_c \in K_c$, $k_f \in K_f - K_c$, $k_b \in K_b - K_c$, and $k_r \in K_r$. If the resulting ciphertext does not match the corresponding ciphertext in the pair, the guessed key bits are incorrect. The size of the set T depends on the size of the block cipher and the main master key. For example, if the block cipher size is 64 bits and the master key size is 128 bits, the size of the set T should be 2.

To perform the attack, assume we have d random pairs of data represented as $D=\{(P_0,C_0),...,(P_{d-1},C_{d-1})\}$, with (P_0,C_0) designated as the primary pair. A set of m bit addresses, $X=\{X[0],...,X[m-1]\}$, is selected as the matching points. For the given pair of data (P,C), with $k_f \in K_f$ and $k_b \in K_b$, assume the Boolean functions used in the forward and backward directions are $F_f(P,k_f)$ and $F_b(C,k_b)$, respectively. The forward direction is used to generate the table H. The steps of the attack proceed as follows:

For each $k_c \in K_c$, perform the following steps:

1. Initialize the hash table H, which is indexed by the differentials of all matching point values in the forward direction. Since there are m matching points and $d-1$ differences, the table H contains $2^{m(d-1)}$ indices. Therefore, assuming the cipher behaves similarly to a random function, we expect to have an average of $2^{|K_f|-|K_c|-m(d-1)}$ values for each index in the table H.
2. a. For each $k_f \in K_f - K_c$, and for each P_i, where $0 \leq i \leq d-1$ in $D = \{(P_0, C_0), \ldots, (P_{d-1}, C_{d-1})\}$, compute the values of all Xf_i, where $Xf_i = F_f(P_i, k_c \| k_f)$.
   b. Compute the values $\Delta Xf_{0i} = Xf_0 + Xf_i$ for $1 \leq i \leq d-1$. Using the obtained value, update the corresponding index in the table H with $k_c \| k_f$.
3. a. For each $k_b \in K_b - K_c$, and for each C_i, where $0 \leq i \leq d-1$ in $D = \{(P_0, C_0), \ldots, (P_{d-1}, C_{d-1})\}$, compute the values of all Xb_i, where $Xb_i = F_b(C_i, k_c \| k_b)$.
   b. Compute the values $\Delta Xb_{0i} = Xb_0 + Xb_i$ for $1 \leq i \leq d-1$, and search for a match for $\Delta Xb_{01} \| \ldots \| \Delta Xb_{0d-1}$ in the table H.
   c. i. For each value k_f in $H[\Delta Xb_{01} \| \ldots \| \Delta Xb_{0d-1}]$ proceed to the following steps.
      ii. A. For each $k_r \in K_r$, set a Flag to true, and perform the following steps.
      B. For each (P'_t, C'_t) in the set T, encrypt the plaintext P'_t using the guessed key bits $k_c \| k_f \| k_b \| k_r$ as $C_t = F(P'_t, k_c \| k_f \| k_b \| k_r)$. If $C_t \neq C'_t$, set the Flag to false. If the Flag is false, this indicates that the guessed key is incorrect, and a different k_r should be tried. In this case, return to step A.
      C. If the Flag remains true, return the guessed key $k_c \| k_f \| k_b \| k_r$ as the master key.

Based on the aforementioned algorithm, where the forward direction is selected to generate the table H, the time complexity of the attack is given by:

$$2^{|K_c|}\left(d \cdot 2^{|K_f|-|K_c|} + 2^{|K_b|-|K_c|} \cdot \left(d + 2^{|K_f|-|K_c|-(d-1)m} \cdot 2^{|K_r|} \cdot t\right)\right) \tag{2}$$

The time complexity formula in Equation (2) can be simplified as follows:

$$\tau = d \cdot 2^{|K_f|} + d \cdot 2^{|K_b|} + t \cdot 2^{|K_f|+|K_b|+|K_r|-|K_c|-(d-1)m} \tag{3}$$

As seen in Equation (3), all terms are precomputed and fixed except for the data complexity d. Therefore, to achieve optimal time and data complexity, Equation (3) can be regarded as a function with one variable d, where the goal is to find the minimum value of d such that $\tau < 2^{|K|}$. The obtained d should then be rounded to the nearest integer.

Based on the algorithm, the memory complexity can be calculated as follows:

$$\min(2^{|K_f|-|K_c|}, 2^{|K_b|-|K_c|}) \tag{4}$$

The process of the algorithm is summarized in Algorithm 1.

Algorithm 1: Our meet in the middle algorithm

Input : d random pairs of data as $D=\{(P_0,C_0),...,(P_{d-1},C_{d-1})\}$,
The bit addresses $X[0],...,X[m-1]$ as the chosen matching points,
t extra pairs of data as $T=\{(P'_0,C'_0),...,(P'_{t-1},C'_{t-1})\}$
Output A value for the key bits in the set $\mathcal{K}$ as candidate
:

```
1  for each k_c in K_c do
2      Initialize the hash table H indexed by
         the differences of the values at the matching points in the forward direction;
3      for each k_f in K_f − K_c do
4          Xf_i = F_f(P_i, k_c‖k_f),   0 ≤ i ≤ d−1;
5          ΔXf_0i = Xf_0 ⊕ Xf_i,   1 ≤ i ≤ d−1 ;
6          H[ΔXf_01‖...‖ΔXf_0d−1] ← H[ΔXf_01‖...‖ΔXf_0d−1] ∪ {k_f} ;
7      for each k_b in K_b − K_c do
8          Xb_i = F_b(C_i, k_c‖k_b),   0 ≤ i ≤ d−1;
9          ΔXb_0i = Xb_0 ⊕ Xb_i,   1 ≤ i ≤ d−1 ;
10         for each k_f in H[ΔXb_01‖...‖ΔXb_0d−1] do
11             for each k_r in K_r do
12                 Flag = True;
13                 for each (P'_t, C'_t) in T do
14                     C_t = F(P'_t, k_c‖k_f‖k_b‖k_r);
15                     if C_t ≠ C'_t then
16                         Flag = False;
17                         Go back to line 11
18                 if Flag is True then
19                     Return k_c‖k_f‖k_b‖k_r as the master key;
```

3 Overview of Our Automatic Tool

In this section, we provide details of our modeling method. As discussed in Section 2, the set of key bits that are nonlinearly involved at the bit addresses determine the complexity. Therefore, the key question is how to calculate the sets of linearly and nonlinearly dependent key bits for each bit address and determine which bits are suitable to be chosen as matching points. The answers to these questions are elaborated below.

3.1 Modeling the Operations

In our technique, all key bits in the sets of linearly and nonlinearly dependent bits for each bit address, whether in the forward or backward direction, must be determined. It is important to note that the intersection of the linearly and nonlinearly dependent sets for each direction in each bit address must be empty.

The information regarding each bit address in the internal rounds, which includes the set of nonlinearly and linearly dependent key bits, relies on the boolean functions $F_i:\mathbb{F}_2^n\rightarrow\mathbb{F}_2$ in $F=(F_0,F_1,...,F_{n-1})$ applied to the corresponding bit. Let $x=(x_0,x_1,...,x_{n-1})$ represent the input bits of F, and $x'=(x'_0,x'_1,...,x'_{n-1})$ represent the corresponding output bits of F. Let the associated bit addresses for the input and output vectors be $(X[0],...,X[n-1])$ and $(X'[0],...,X'[n-1])$ respectively.

Each bit address $X[i]$ ($X'[i]$), where $i \in \{0,1,...,n-1\}$, is a tuple (N_i,L_i) (N'_i,L'_i), where N_i (N'_i) and L_i (L'_i) denote the sets of nonlinearly and linearly dependent key bits, respectively, and $N_i \subseteq \boldsymbol{\kappa}$ $(N'_i \subseteq \boldsymbol{\kappa})$ and $L_i \subseteq \boldsymbol{\kappa}$ $(L'_i \subseteq \boldsymbol{\kappa})$, with $N_i \cap L_i = \emptyset$ $(N'_i \cap L'_i = \emptyset)$.

Assume that all of the tuples (N_i,L_i), where $i \in \{0,1,...,n-1\}$, for the bit addresses $X[i]$ are already determined, and we aim to calculate the tuples (N'_i,L'_i) for the bit addresses $X'[i]$. For each bit address $X'[i]$, which is the output of the function F_i, we define two sets of dependency indices I_i and J_i, where $I_i \cap J_i = \emptyset$, based on certain properties of F_i (explained in the following). The set I_i contains the indices of the bit addresses $X[u]$ $(u \in \mathbb{Z}_n)$ on which $X'[i]$ nonlinearly depends, and similarly, J_i contains the indices of the bit addresses $X[v]$ $(v \in \mathbb{Z}_n)$ on which $X'[i]$ linearly depends. Therefore, we have:

$$\begin{aligned} &I_i = \{u \,|\, u \in \mathbb{Z}_n \text{ and } X'[i] \text{ nonlinearly depends on } X[u]\} \\ &J_i = \{v \,|\, v \in \mathbb{Z}_n \text{ and } X'[i] \text{ linearly depends on } X[v]\} \\ &I_i \cap J_i = \emptyset \end{aligned} \tag{5}$$

Given the algebraic normal form of F_i, where $i \in \{0,1,...,n-1\}$, the dependency sets I_i and J_i for the given bit address $X'[i]$ can be calculated. To clarify, consider the block cipher PRESENT [BKL+07] as an example. In PRESENT, if the vector (x_3,x_2,x_1,x_0) represents the input and the vector (x'_3,x'_2,x'_1,x'_0) represents the output of the S-box function, the algebraic normal form of the PRESENT S-box is expressed as follows:

$$\begin{aligned} x'_3 &= x_1x_2 + x_0 + x_2 + x_3 \\ x'_2 &= x_0x_1x_2 + x_0x_1x_3 + x_0x_2x_3 + x_1x_3 + x_2x_3 + x_1 + x_3 \\ x'_1 &= x_0x_1x_3 + x_0x_2x_3 + x_0x_1 + x_0x_3 + x_1x_3 + x_2 + x_3 + 1 \\ x'_0 &= x_0x_1x_2 + x_0x_1x_3 + x_0x_2x_3 + x_1x_2 + x_0 + x_1 + x_3 + 1 \end{aligned} \tag{6}$$

As shown in Equation (6), the first (x'_0), second (x'_1), and third (x'_2) output bits nonlinearly depend on all four input bits. However, the fourth (x'_3) output bit nonlinearly depends on the second and third input bits and linearly depends on the first and fourth input bits. Therefore, the corresponding sets of dependencies I_i and $J_i, 0 \leq i \leq 3$ can be represented as follows:

$$\begin{aligned} &I_0 = I_1 = I_2 = \{0,1,2,3\}, \quad J_0 = J_1 = J_2 = \emptyset \\ &I_3 = \{1,2\} \quad J_3 = \{0,3\} \end{aligned} \tag{7}$$

Without loss of generality, to compute (N'_i,L'_i) for $i \in \{0,...,n-1\}$ at bit address $X'[i]$, we define two functions N_{F_i} and L_{F_i}. The function N_{F_i} (L_{F_i}) is designed to establish N'_i (L'_i). N_{F_i} takes as input a set of bit addresses at the input of F_i, indexed by I_i and J_i, and returns a subset of $\boldsymbol{\kappa}$. On the other hand, L_{F_i} only takes as input a set of bit addresses indexed by J_i and returns a subset of $\boldsymbol{\kappa}$. It is essential to note that the intersection of the outputs of N_{F_i} and L_{F_i} must be empty. Therefore, N_{F_i} and L_{F_i} are formulated as follows:

$$
\begin{aligned}
&N_{F_i}:(\prod_{j\in I_i}(N_j,L_j)\times\prod_{j\in J_i}(N_j,L_j))\to\mathcal{P}(\mathcal{K})\\
&N_{F_i}(\{X[j]\,|\,j\in I_i\},\{X[j]\,|\,j\in J_i\})\to\bigcup_{j\in I_i}(N_j\cup L_j)\cup\bigcup_{j\in J_i}N_j=N_i'\\
&L_{F_i}:(\prod_{j\in J_i}(N_j,L_j),N_i')\to\mathcal{P}(\mathcal{K})\\
&L_{F_i}(\{X[j]\,|\,j\in J_i\},N_i')\to\bigcup_{j\in J_i}L_j\setminus N_i'=L_i'
\end{aligned}
\tag{8}
$$

As shown in Equation (8), the functions N_{F_i} and L_{F_i} are used to determine the dependency of each key bit at each bit address $X'[i]$. Therefore, for the S-box of the PRESENT, if we assume that the bit addresses $X[i]=(N_i,L_i),0\le i\le 3$ correspond to the vector (x_3,x_2,x_1,x_0) as the input of the S-box, for calculating the data in the bit addresses $X'[i]$, we have:

$$
\begin{aligned}
&N_u'=\bigcup_{i\in I_u}(N_i\cup L_i),\quad L_u'=\emptyset\quad u\in\{0,1,2\}\\
&N_3'=\bigcup_{i\in I_3}(N_i\cup L_i)\cup\bigcup_{i\in J_3}N_i\quad L_3'=\bigcup_{i\in J_3}L_i\setminus N_3'
\end{aligned}
\tag{9}
$$

It is noteworthy that linear operations in cryptographic primitives, such as MixColumn and `XOR`, can be represented using the same formula introduced in Equation (8). However, it should be noted that the dependency sets I_i, where $i\in\{0,1,...,n-1\}$, are $\emptyset$ for these operations. Before proceeding to the next section, we introduce the following definition:

Definition 1. ***Permissible matching points:*** Permissible matching points are those bits whose corresponding bit addresses contain information that leads to a successful attack.

3.2 Algorithm to Find the Matching Points

In this section, we elucidate the functionality of our automated tool for implementing MITM attacks. Specifically, we delve into computing the time, data, and memory complexities and also the process of selecting matching points. To illustrate this, we refer to Figure 2, which depicts the structure of a cipher.

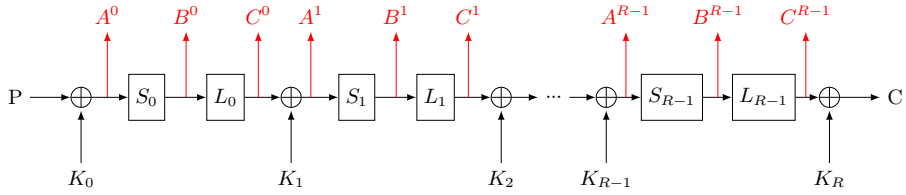

Fig. 2: Illustration of a cipher, where the positions $A^r[i]$, $B^r[i]$, and $C^r[i]$ (for $0\le r<R$ and $0\le i<n$) are designated as the bit addresses.

In Figure 2, assume that the cipher has a block size of n and comprises R rounds. The round keys K_i, $0\le i\le R$, are derived using the key schedule algorithm from the

set of all key bits $\mathcal{K}$. Our goal is to target the maximum number of rounds feasible, while ensuring that the time complexity remains lower than exhaustive search.

To model the cipher with our tool, we first determine which positions are suitable to be considered as bit addresses. Each position is selected based on the operations applied in the cipher. Specifically, positions after operations that modify the arrangement of nonlinearly dependent key bits are ideal candidates for bit addresses. In Figure 2, the positions $A^r[i]$, $B^r[i]$, and $C^r[i]$, where $0 \leq r < R$ and $0 \leq i < n$, are chosen as the bit addresses, and each bit address is modeled according to the method described in Section 3.1. For each bit address, the tool then computes the sets of linearly and nonlinearly dependent key bits in both the forward and backward directions. Once these sets are established, the cardinality of each set of nonlinearly dependent key bits is calculated based on the key schedule [2]. This enables the computation of the time complexity required to recover the key bits at each bit address.

It is evident that as the number of rounds increases, the size of the set of nonlinearly dependent key bits in both the forward and backward directions also increases, until it reaches a point where the time complexity for all bit addresses becomes greater than or equal to $2^{|\mathcal{K}|}$. This indicates that the attack is no longer faster than an exhaustive search, and no permissible matching points can be found among the bit addresses. Therefore, to find the maximum number of rounds possible to perform the attack, the number of rounds R is increased as long as the time complexity remains lower than $2^{|\mathcal{K}|}$.

The following provides an explanation of how our automatic tool operates. For simplicity, we define the following terms for a given bit address $X^r[i]$:

$XK_f^r[i]$	The set of nonlinearly dependent key bits in the forward direction.
$XK_b^r[i]$	The set of nonlinearly dependent key bits in the backward direction.
$XK_u^r[i]$	$XK_u^r[i] = XK_f^r[i] \cup XK_b^r[i]$.
$XK_c^r[i]$	$XK_c^r[i] = XK_f^r[i] \cap XK_b^r[i]$.
$XK_r^r[i]$	$XK_r^r[i] = \mathcal{K} - XK_u^r[i]$.
$XT^r[i]$	the time complexity to recover the master key if $X^r[i]$ is a matching point.
$XD^r[i]$	the data complexity to recover the master key if $X^r[i]$ is a matching point.
$XM^r[i]$	the memory complexity to recover the master key if $X^r[i]$ is a matching point.
$XT'^r[i]$	$XT'^r[i] = \max\big(\lvert XK_f^r[i]\rvert, \lvert XK_b^r[i]\rvert\big)$.
$XM'^r[i]$	$XM'^r[i] = \min\big(\lvert XK_f^r[i]\rvert, \lvert XK_b^r[i]\rvert\big)$.

For each bit address $X^r[i]$, our automatic tool first forms the sets $XK_f^r[i]$ and $XK_b^r[i]$, and then derives the values $|XK_f^r[i]|$, $|XK_b^r[i]|$, $|XK_c^r[i]|$, and $|XK_r^r[i]|$ from these sets. According to relations 3 and 4, $XT'^r[i]$ determines the time complexity, while $XM'^r[i]$ determines the memory complexity.

After calculating $XT'^r[i]$ for each bit address $X^r[i]$, the bit addresses with the minimum value of $XT'^r[i]$ are selected as candidates for the matching points. If this minimum value equals $|\mathcal{K}|$, it indicates that no permissible matching point exists, and the attack is no longer faster than exhaustive search.

By using the values $XM'^r[i]$ and $XK_c^r[i]$ for each bit address $X^r[i]$ selected as a candidate, the memory complexity $XM^r[i]$ can be calculated. Therefore, among these

[2] There may exist dependencies among the key-bits due to the key schedule. For instance, refer to the details of PRESENT80 in Section 4.4.

candidates, those with the minimum $XM^r[i]$ value are selected as the new candidates for the matching points, and the minimum value is returned as the memory complexity.

According to Equation (3), to achieve better time and data complexity, the number of matching points m should be as large as possible. Therefore, among the list of candidates, those that share the same set of nonlinearly dependent key bits in the forward direction and also same set of nonlinearly dependent key bits in the backward direction are grouped into distinct sets. Among these sets, the one with the maximum size is randomly selected as the final set of candidates for the matching points. It should be noted that the size of this final set represents the number of matching points m.

At this stage, all values $|K_f|$,$|K_b|$,$|K_c|$,$|K_r|$, and m have been computed, and according to Equation (3), the value of the function τ depends only on the variable d. Therefore, the data complexity d is determined to minimize the time complexity τ. If the obtained value is greater than or equal to $2^{|\boldsymbol{\kappa}|}$, it indicates that there is no permissible matching point for the given round R. Algorithm 2, summarizes the functioning of our automatic tool.

Algorithm 2: The algorithm used in our automatic tool.

Input : Given cipher with size n and round R, with the set of its master key $\boldsymbol{\kappa}$ and its key schedule

Output : Time complexity T, Memory complexity M, Data complexity D, set of bit addresses as the matching points

for *each* r *in* R **do**
- -Model each $X^r[i]$ and form the sets $XK_f^r[i]$ and $XK_b^r[i]$;
- -For each $X^r[i]$, derive the values $|XK_f^r[i]|$,$|XK_b^r[i]|$,$|XK_c^r[i]|$,$|XK_r^r[i]|$,$XT'^r[i]$ and $XM'^r[i]$.;
- -Select the bit addresses that have minimum value for $XT'^r[i]$ as candidates and name the minimum value T';
- **if** $T'=|\boldsymbol{\kappa}|$ **then**
  - Terminate the algorithm;
- **else**
  - -For each $X^r[i]$ in the list of candidates, calculate $XM^r[i]$;
  - -Update the list of candidates by selecting the bit addresses that have minimum value for $XM^r[i]$ and name the minimum value M;
  - -**Return** M as the memory complexity;
  - -For the candidates, categorize them, by inserting those that share the same set of keys in the forward and backward direction in a set;
  - -Name the maximum size of the sets m (number of the matching points);
  - -Randomly select one of the sets with the size of m and name the set S;
  - -Minimize the function τ, by carefully selecting a value for d, in the Equation 3;
  - -Name the obtained value for τ, T and name d as D;
  - **if** $T\geq 2^{|\boldsymbol{\kappa}|}$ **then**
    - Terminate the algorithm;
  - **else**
    - **Return** T as the time complexity;
    - **Return** D as the data complexity;
    - **Return** S as the set of matching points;

4 Applications

In this section, we illustrate how our tool can be applied to various lightweight block ciphers. Specifically, for the ciphers CRAFT [BLMR19] and the Midori family [BBI+15], which employ MixColumn in their design, we introduce two attack strategies. The first approach is the "regular technique", while the second leverages what we term the "equivalent key technique."

In the equivalent key technique, during backward analysis, we do not guess a key bit in its original position post-MixColumn. Instead, we guess an equivalent key bit before the MixColumn, where the MixColumn establishes a linear relationship between the key bits and their equivalent counterparts. This method, initially introduced by [IS13], was developed to perform key recovery attacks on Feistel structures, aiming to reduce the number of guessed key bits by substituting original key bits with equivalent ones. By applying the equivalent key technique to CRAFT, we were able to enhance the key recovery attack; however, no improvement was observed for the Midori family.

All experiments were conducted on a personal computer running Ubuntu 22.04.3 LTS, equipped with an Intel Core i5-10210U CPU (4 cores, 2 threads) and 16 GB of RAM. The code was implemented in Python 3.10. The tool demonstrated high computational efficiency: for example, the CRAFT regular technique required approximately 3 seconds to complete, while the equivalent technique took 52 seconds.

4.1 Application to CRAFT

CRAFT is a lightweight tweakable SPN block cipher devised by Beierle et al. In this section, we present our results on CRAFT. We conducted regular and equivalent key attacks on CRAFT. With the regular attack, we successfully attacked to 14 rounds, while employing the equivalent key technique enabled us to reach 15 rounds.

4.1.1 Specification of CRAFT

CRAFT employs a 64-bit block, a 128-bit key, and a 64-bit tweak. The state is organized as a 4×4 matrix of nibbles, represented by $I_{i,j}$, where i and j denote the row and column positions, respectively. This matrix can also be seen as a vector by concatenating the rows.

The 128-bit key K is split into two 64-bit subkeys, K_0 and K_1. Along with the 64-bit tweak T, these are used to derive four 64-bit tweakeys, namely TK_0, TK_1, TK_2, and TK_3. After initializing the state with the plaintext, the cipher performs 31 round functions ($R_i, 0 \leq i \leq 30$) followed by an additional linear round R'_{31} to produce the ciphertext. Each round function R_i includes five operations: *SubBox* (SB), *MixColumn* (MC), *PermuteNibbles* (PN), *AddConstant*$_i$ (ARC_i), and *AddTweakey*$_i$ (ATK_i). However, R'_{31} only consists of MC, ARC_i, and ATK_i operations. The definitions for the round functions $R_i, i \in \{0,...,30\}$ and R'_{31} are as follows:

$$\begin{aligned} R_i &= SB \circ PN \circ ATK_i \circ ARC_i \circ MC \\ R'_{31} &= ATK_{31} \circ ARC_{31} \circ MC \end{aligned} \tag{10}$$

In the following, we elaborate on the operations. Since our model does not depend on ATK_i and ARC_i, we omit the details of those functions.

SB: The 4-bit involutory S-box S is simultaneously applied 16 times, i.e., to each nibble of the state. This S-box mirrors the one used in the block cipher MIDORI [BBI+15] and WARP [BBI+20]. The table for the S-box (in hexadecimal notation) is provided in Table 2.

Table 2: The S-box of MIDORI, WARP and CRAFT

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	A	D	3	E	B	F	7	8	9	1	5	0	2	4	6

PN: An involutory permutation P is applied to the nibble positions of the state. Specifically, for all $0 \leq i \leq 15$, I_i is replaced by $I_{P(i)}$, where

$$P = [15,12,13,14,10,9,8,11,6,5,4,7,1,2,3,0]$$

MC: The following involutory binary matrix M is multiplied to each column of the state:

$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For each column index $j \in \{0,1,2,3\}$,

$$\begin{bmatrix} I_{0,j} \\ I_{1,j} \\ I_{2,j} \\ I_{3,j} \end{bmatrix} \rightarrow \begin{bmatrix} I_{0,j} \oplus I_{2,j} \oplus I_{3,j} \\ I_{1,j} \oplus I_{3,j} \\ I_{2,j} \\ I_{3,j} \end{bmatrix}$$

Key Schedule: The 128-bit master key is divided into two 64-bit segments, $K0$ and $K1$. $K0$ is applied during even rounds, while $K1$ is employed in the odd rounds.

4.1.2 Key Recovery Attack on CRAFT

In order to utilize our tool, the initial step involves determining the bits within the round functions that are to be regarded as the bit addresses, along with describing the operations at the bit level. Figure 3 illustrates the CRAFT structure and highlights the positions designated as the bit addresses in the regular technique.

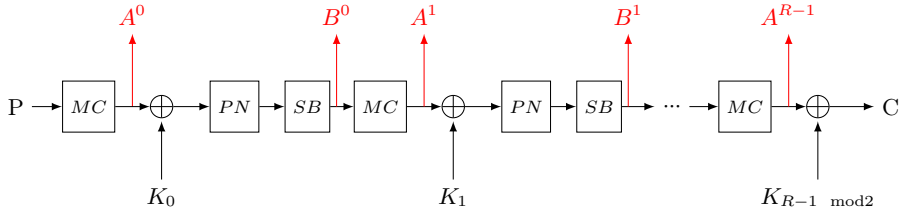

Fig. 3: Regular attack on CRAFT. The positions $A^r, B^r, 0 \leq r < R$ are considered as the bit Addresses.

Next, to illustrate the S-box, we must establish the dependency sets as described in Section 3.1. Based on the structure of the CRAFT S-box, the dependency sets are as follows:

$$\begin{aligned} &I_0 = I_1 = I_3 = \{0,1,2,3\}, \quad J_0 = J_1 = J_3 = \emptyset \\ &I_2 = \{0,1,3\} \quad J_2 = \emptyset \end{aligned} \tag{11}$$

Since the S-box is involutory, the dependency sets for the forward and backward directions remain the same.

We represent the set of all key bits as $\boldsymbol{\kappa} = \{k_0,...,k_{127}\}$. The round keys K_0 and K_1, derived from $\boldsymbol{\kappa}$ based on the key schedule and used alternately in the round functions of CRAFT, are as follows:

$$\begin{aligned} K_0 &= \{k_0,k_1,...,k_{63}\} \\ K_1 &= \{k_{64},k_{65},...,k_{127}\} \end{aligned} \tag{12}$$

To clarify how our automatic tool works, based on Figure 3, the method by which the sets of nonlinearly dependent key bits $XN^r[i]$, where $0 \leq i < 64$ and $0 \leq r < 32$, and linearly dependent key bits $XL^r[i]$ for a given bit address $X^r[i]$ are formed is explained in Table 3. The process begins by forming the sets at bit addresses $A^0[i]$ in the forward direction and $A^{r-1}[i]$ in the backward direction, continuing until the set of all nonlinearly dependent key bits for all bit addresses equals $|\boldsymbol{\kappa}|$.

Table 3: The process by which the set of key bits is formed in CRAFT in the forward and backward directions.

$A^r[0]$	$A^r[1]$	$A^r[2]$	...	$A^r[63]$
Forward Direction				
$AN^0[0] = \emptyset$ $AL^0[0] = \emptyset$	$AN^0[1] = \emptyset$ $AL^0[1] = \emptyset$	$AN^0[2] = \emptyset$ $AL^0[2] = \emptyset$	$\cdots$	$AN^0[63] = \emptyset$ $AL^0[63] = \emptyset$
$BN^0[0] = \{k_{60},k_{61},k_{62},k_{63}\}$ $BL^0[0] = \emptyset$	$BN^0[1] = \{k_{60},k_{61},k_{62},k_{63}\}$ $BL^0[1] = \emptyset$	$BN^0[2] = \{k_{60},k_{62},k_{63}\}$ $BL^0[2] = \emptyset$	$\cdots$	$BN^0[63] = \{k_0,k_1,k_2,k_3\}$ $BL^0[63] = \emptyset$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$AN^8[0] = \boldsymbol{\kappa}$ $AL^8[0] = \emptyset$	$AN^8[1] = \boldsymbol{\kappa}$ $AL^8[1] = \emptyset$	$AN^8[2] = \boldsymbol{\kappa}$ $AL^8[2] = \emptyset$	$\cdots$	$AN^8[63] = \boldsymbol{\kappa}$ $AL^8[63] = \emptyset$
Backward Direction				
$AN^{r-1}[0] = \emptyset$ $AL^{r-1}[0] = k_{64}$	$AN^{r-1}[1] = \emptyset$ $AL^{r-1}[1] = k_{65}$	$AN^{r-1}[2] = \emptyset$ $AL^{r-1}[2] = k_{66}$	$\cdots$	$AN^{r-1}[63] = \emptyset$ $AL^{r-1}[63] = k_{127}$
$BN^{r-2}[0] = \emptyset$ $BL^{r-2}[0] = \{k_{64},k_{96},k_{112}\}$	$BN^{r-2}[1] = \emptyset$ $BL^{r-2}[1] = \{k_{65},k_{97},k_{113}\}$	$BN^{r-2}[2] = \emptyset$ $BL^{r-2}[2] = \{k_{66},k_{98},k_{114}\}$	$\cdots$	$BN^{r-2}[63] = \emptyset$ $BL^{r-2}[63] = \{k_{127}\}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$BN^{r-9}[0] = \boldsymbol{\kappa}$ $BL^{r-9}[0] = \emptyset$	$BN^{r-9}[1] = \boldsymbol{\kappa}$ $BL^{r-9}[1] = \emptyset$	$BN^{r-9}[2] = \boldsymbol{\kappa}$ $BL^{r-9}[2] = \emptyset$	$\cdots$	$BN^{r-9}[63] = \boldsymbol{\kappa}$ $BL^{r-9}[63] = \emptyset$

In the regular attack, we managed to attack 14 rounds with time, data, and memory complexities of $2^{126.5}$, 3 and 2^4, respectively. The bit addresses identified as matches by our tool are $A^6[16]$,$A^6[17]$,$A^6[18]$ and $A^6[19]$. The parameters obtained using our tool are $K_f = K_b = 124$, $K_u = 128$, $K_c = 120$, $K_r = 0$, and $m = 4$. According to the structure of the cipher, we have $t = 2$.

In the equivalent technique case, the positions chosen as bit addresses are shown in Figure 4.

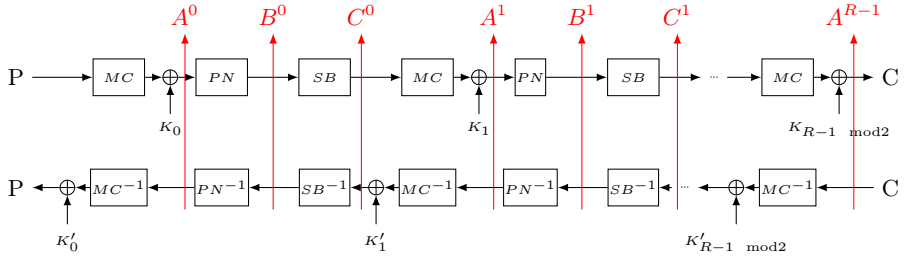

Fig. 4: Equivalent key attack on CRAFT. the positions A^r,B^r,C^r, $0 \leq i < R$ are considered as the bit Addresses. In the backward direction the key addition is moved prior to Mixcolumn

As explained earlier, in the equivalent key technique, the round keys in the backward direction are shifted prior to the MixColumn operation. As a result, the round keys are represented as $K'_0 = \{k'_0,...,k'_{63}\}$ and $K'_1 = \{k'_{64},...,k'_{127}\}$, where each k'_i $(0 \leq i \leq 127)$ is obtained by applying the inverse of the MixColumn operation to the original k_i values[3]. Consequently, each k'_i is a linear combination of the original k_i values. Specifically, based on the structure of the CRAFT MixColumn, we have:

$$\begin{aligned} k'_0 &= k_0 + k_{32} + k_{48} \\ k'_1 &= k_1 + k_{33} + k_{49} \\ k'_2 &= k_2 + k_{34} + k_{50} \\ &\vdots \end{aligned} \tag{13}$$

Based on the output of the tool, we successfully attacked 15 rounds with time, data and memory complexities of $2^{126.5}$, 3 and 2^4, respectively. The bit addresses identified as the matching points by our tool are$A^6[16]$,$A^6[17]$,$A^6[18]$ and $A^6[19]$. The parameters are the same as those derived in the regular attack.

To the best of our knowledge, there is one other reported MITM attack on CRAFT, proposed by [MLC23]. Their attack targets 20 rounds with time, data and memory complexities of $2^{126.94}$, 2^{56} and 2^{109}, respectively. Although their attack covers more rounds, it is based on a chosen-plaintext attack and requires significantly more data and memory compared to our approach.

4.2 Application to Midori

Midori is an SPN variant designed by Banik et al. It comprises two block ciphers: Midori64 and Midori128. In this section, we detail our results on the Midori family. In both th regular and equivalent key technique, we successfully attacked 5 and 7 rounds for Midori64 and Midori128, respectively.

[3] Since the MixColumn of CRAFT is involutory, the MixColumn and its inverse are identical.

4.2.1 Specification of Midori

Both Midori ciphers utilize 128-bit keys but differ in block size, with Midori64 employing $n=64$ and Midori128 using $n=128$. Midori64 comprises 16 rounds, while Midori128 consists of 20 rounds.

Midori adopts an SPN variant architecture, characterized by the S-layer and the P-layer, utilizing a 4×4 array called the state for data representation. The cell sizes m are 4 and 8 bits for Midori64 and Midori128, respectively.

The round function of Midori comprises an S-layer SubCell, a P-layer ShuffleCell, and MixColumn. Each layer updates an n-bit state S. For detailed operations, refer to [BBI+15]. The round functions of Midori are outlined in the following algorithm.

Algorithm 3: Midori Description

Input : $X, WK, RK_0, ..., RK_{R-2}$
Output Y
:
$S \longleftarrow KeyAdd(X, WK)$;
for $I=0$ *to* $R-2$ **do**
  $S \longleftarrow SubCell(S)$;
  $S \longleftarrow ShuffleCell(S)$;
  $S \longleftarrow MixColumn(S)$;
  $S \longleftarrow KeyAdd(S, RK_i)$;
$S \longleftarrow SubCell(S)$;
$Y \longleftarrow KeyAdd(S, WK)$;

Key Schedule: Given that we do not include the addition of any constant, for Midori64, a 128-bit secret key K is represented as two 64-bit keys, K_0 and K_1, concatenated as $K = K_0 \| K_1$. Then, $WK = K_0 \oplus K_1$, and $RK_i = K_{(i \mod 2)}$ where $0 \le i \le 14$. For Midori128, $WK = K$ and $RK_i = K$ for $0 \le i \le 18$.

4.2.2 Key Recovery Attack on Midori

Figure 5 illustrates the structure of Midori along with the designated bit address positions utilized in the regular approach.

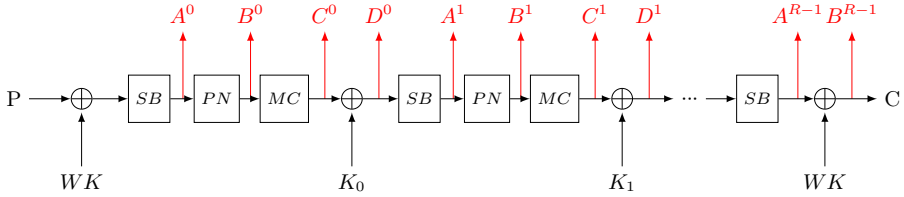

Fig. 5: Regular attack on Midori. the positions A^r, B^r, C^r, D^r, $0 \le r < R$ are considered as the bit Addresses.

The S-box utilized in Midori64 is identical to CRAFT. However, for Midori128, a different S-box is used and the dependency sets for its S-box are as follows:

$$I_0 = I_1 = I_2 = I_3 = \{0,1,2,3\}, \quad J_0 = J_1 = J_2 = J_3 = \emptyset \tag{14}$$

If we represent the set of all key bits as $\mathcal{K}=\{k_0,...,k_{127}\}$. The round keys WK, K_0 and K_1, derived from $\mathcal{K}$ based on the key schedule and used alternately in the round functions of Midori64, are as follows:

$$\begin{aligned} WK &= \{k_0+k_{64}, k_1+k_{65}, ..., k_{63}+k_{127}\} \\ K_0 &= \{k_0, k_1, ..., k_{63}\} \\ K_1 &= \{k_{64}, k_{65}, ..., k_{127}\} \end{aligned} \tag{15}$$

For Midori128 we have:

$$WK = K_0 = K_1 = \{k_0, k_1, ..., k_{127}\} \tag{16}$$

For Midori64 in the regular technique, we were able to attack 5 rounds with the time, data and memory complexity of 2^{77},16 and 2^{48}, respectively, with matching points are $C^1[20]$,$C^1[21]$,$C^1[22]$ and $C^1[23]$. Using our tool, we derived the following parameters: $K_f = K_b = 72$, $K_u = 120$, $K_c = 24$, $K_r = 8$, and $m = 4$. Based on the cipher's structure, this implies $t=2$.

For Midori128, we were able to attack 7 rounds with the time, data and memory complexity of $2^{126.3}, 3, 2^0$, respectively, with the matching points are $C^2[40]$,$C^2[43]$,$C^2[45]$ and $C^2[46]$. Our tool yields the parameter values $K_f = K_b = 124$, $K_u = 124$, $K_c = 124$, $K_r = 4$, and $m = 4$. According to the cipher design, the corresponding value of t is 1.

For the equivalent key technique, we used Figure 6 for reference.

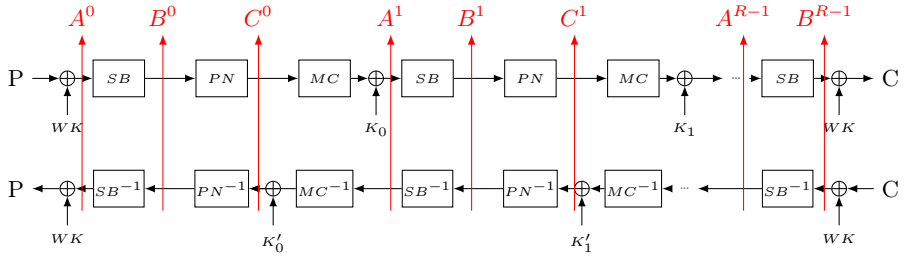

Fig. 6: Equivalent key attack on Midori. the positions $A^r, B^r, C^r, \quad 0 \leq r < R$ are considered as the bit Addresses. In the backward direction the key addition is moved prior to Mixcolumn.

For Midori64, we were able to attack 5 rounds with the time, data and memory complexity of $2^{86.6}$,49 and 2^{39} respectively. The bit address that our tool returned as the matching point is $B^2[6]$. The analysis performed by our tool produces the parameters $K_f=76$, $K_b=81$, $K_u=120$, $K_c=37$, $K_r=8$, and $m=1$, with the cipher's structure indicating that $t=2$.

For Midori128, we were able to attack 6 rounds with the time, data and memory complexity of $2^{125.5}, 3$ and 2^4, respectively. The matching points are $A^3[32]$, $A^3[34]$, $A^3[37]$ and $A^3[39]$. According to the results from our tool, the parameters are set as $K_f=124$, $K_b=88$, $K_u=128$, $K_c=84$, $K_r=0$, and $m=4$. The structure of the cipher further determines $t=1$.

As far as we know, only one other MITM attack on Midori64 has been documented, as introduced by [LW17]. This attack focuses on 12 rounds and has time, data and memory complexities of $2^{125.5}$, $2^{55.5}$ and 2^{106}, respectively.

4.3 Application to WARP

The lightweight block cipher WARP, proposed by Banik et al. [BBI+20], is a variant of the 32-nibble Type-2 Generalized Feistel Network (GFN). In this section, we present our results on WARP. We were able to attack 18 rounds of WARP.

4.3.1 Specification of WARP

WARP is a 128-bit block cipher with a 128-bit key. Its structure is a variant of the 32-branch Type-II GFN. During encryption and decryption processes, a 128-bit plaintext M and a ciphertext C are loaded into a 128-bit internal state, respectively. This internal state is represented as 32 nibbles, $X = X_0 \parallel X_1 \parallel \cdots \parallel X_{31}$, where $X_i \in \{0,1\}^4$. The 128-bit secret key K is composed of two 64-bit keys K^0 and K^1, denoted as $K = K^0 \parallel K^1$, where $K^i \in \{0,1\}^{64}$.

The round function of WARP involves a 4-bit S-box S, a nibble `XOR`, and a shuffle operation π applied to 32 nibbles. This function undergoes a non-linear transformation that includes a single evaluation of S and round-key addition for each pair of consecutive nibbles. Additionally, it incorporates the addition of a round constant and the application of π to all 32 nibbles. The S-box utilized in WARP is identical to that of CRAFT and Midori.

The WARP cipher consists of 41 rounds, with the nibble shuffle operation π omitted in the final round. Each round, indexed from $i = 1$ to $i = 41$, utilizes a 64-bit (16 nibbles) round key RK^i. The i-th round key RK^i is defined as $RK^i = K^{(i-1) \bmod 2}$ identical to CRAFT.

4.3.2 Key Recovery Attack on WARP

The S-box utilized in WARP is identical to those used in Midori and CRAFT. As WARP's structure does not incorporate MixColumn operations, we exclusively executed a regular attack. The round keys K_0 and K_1 used in WARP are as the same as used in CRAFT 12.

Leveraging our tool, we successfully attacked 18 rounds with time, data and memory complexities of 2^{125}, 7, and 2^6, respectively. The matching point is the single bit $B^8[2]$. Our tool computes the parameters as follows: $K_f = 122$, $K_b = 118$, $K_u = 128$, $K_c = 112$, $K_r = 0$, and $m = 1$. From the structural characteristics of the cipher, it follows that $t = 1$.

4.4 Application to PRESENT80

PRESENT80 is an ultra-lightweight block cipher designed as an SP-network, operating through 31 rounds. It has a block size of 64 bits and supports a key size of 80 bits. Each of the 31 rounds includes an `XOR` operation to add a round key K_r for $1 \leq r \leq 32$, with K_{32} used for post-whitening. Additionally, each round involves a linear bitwise permutation and a nonlinear substitution layer. The nonlinear layer utilizes a single 4-bit S-box S, which is applied 16 times in parallel during each round.

4.4.1 Specification of PRESENT80

Algorithm 4 provides a simple pseudocode description of PRESENT80.

Algorithm 4: PRESENT80 Description

```
generateRoundKeys();
for I=1 to 31 do
    addRoundKey(state,K_i);
    sBoxLayer(state);
    pLayer(state);
addRoundKey(state,K_32);
```

The operations involved in PRESENT80, including the key schedule, are outlined as follows:

addRoundKey. For each round key $K_i = k^i_{63}...k^i_0$ where $1 \leq i \leq 32$ and the current state $b_{63}...b_0$, the addRoundKey operation is defined as: $b_j \oplus k^i_j$ for $0 \leq j \leq 63$.

sBoxLayer. The S-box used in PRESENT80 is a 4-bit to 4-bit substitution box $S : \mathbb{F}_2^4 \longrightarrow \mathbb{F}_2^4$. Its action, represented in hexadecimal notation, is provided in Table 4.

Table 4: The S-box of PRESENT

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

pLayer. The bit permutation applied in PRESENT80 is defined by the following equation. The bit at position i in the state is relocated to position $P(i)$.

$$P(i) = \begin{cases} 16i \bmod 63 & \text{for } 0 \leq i \leq 62 \\ 63 & \text{for } i = 63 \end{cases} \tag{17}$$

Key Schedule. PRESENT80 utilizes an 80-bit key. The key is stored in a key register $\boldsymbol{\kappa}$, denoted as $k_{79}...k_0$. For round r, the 64-bit round key $K_r = \kappa_{63}...\kappa_0$ is derived from the 64 most significant bits of the current state of register $\boldsymbol{\kappa}$. Consequently, at round i, the key is defined as follows:

$$K_r = \kappa_{63},...,\kappa_0 = \{k_{79},...,k_{16}\} \tag{18}$$

After deriving the round key K_r, the key register $\boldsymbol{\kappa} = k_{79}...k_0$ is updated as follows:

$$\begin{aligned} [k_{79}\cdots k_0] &= [k_{18}k_{17}\cdots k_{19}] \\ [k_{79}k_{78}k_{77}k_{76}] &= S[k_{79}k_{78}k_{77}k_{76}] \\ [k_{19}k_{18}k_{17}k_{16}k_{15}] &= S[k_{19}k_{18}k_{17}k_{16}k_{15}] \end{aligned} \tag{19}$$

Thus, the key register is rotated by 61 bit positions to the left, the left-most four bits are passed through the PRESENT S-box. The key register is then shifted left by 61 positions, after which the four most significant bits are processed through the PRESENT S-box.

4.4.2 Key Recovery Attack on PRESENT80

Although the most effective key-recovery attacks on the block cipher PRESENT are based on linear cryptanalysis, they require high data complexity. In contrast, MITM attacks offer the benefit of lower data complexity.

The dependency sets for the S-box of PRESENT80 are defined in Equation 7. However, it should be noted that in the backward direction, the inverse of the S-box is applied. The dependency sets for the backward direction are as follows:

$$\begin{aligned} &I_0 = I_1 = I_2 = \{0,1,2,3\}, \quad J_0 = J_1 = J_2 = \emptyset \\ &I_3 = \{0,2\} \quad J_3 = \{1,3\} \end{aligned} \tag{20}$$

According to the key schedule of PRESENT80, each round of key scheduling applies an S-box, resulting in the definition of 4 new key variables for each round. Therefore, if we represent the set of all key bits as $\mathcal{K} = \{k_0,...,k_{79}\}$, the round keys K_r, $0 \leq r \leq R$, derived from $\mathcal{K}$ based on the key schedule, are as follows:

$$\begin{aligned} K_0 &= \{k_{79},k_{78},k_{77},k_{76},...,k_{16}\} \\ K_1 &= \{k_{83},k_{82},k_{81},k_{80},...,k_{35}\} \\ K_2 &= \{k_{87},k_{86},k_{85},k_{84},...,k_{54}\} \\ &\vdots \end{aligned} \tag{21}$$

The way the round keys are defined creates dependencies among the key bits, making it complex to compute the cardinality of the key bit set for each bit address. To address this issue, we use the Autoguess tool proposed by [HE22]. We form the set of nonlinearly dependent key bits for each bit address, provide it to the Autoguess tool as input, and obtain the cardinality of the set. In Figure 7, we describe the round functions of PRESENT80 and show how the bit addresses are defined.

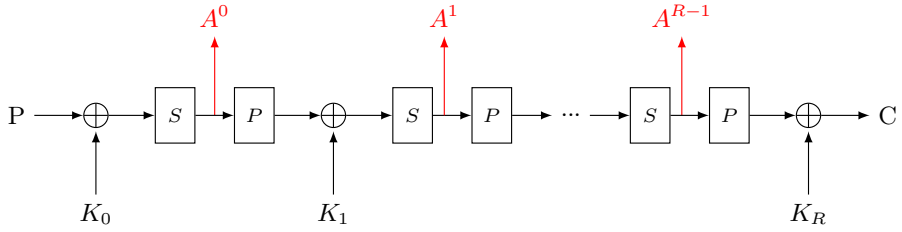

Fig. 7: Attack on PRESENT80. the positions $A^r, \quad 0 \leq r < R$ are considered as the bit Addresses.

For PRESENT80, we were able to attack 6 rounds with time, data and memory complexities of $2^{68.2}$, 18, and 2^{15}, respectively, with the matching point being the bit $A^2[63]$. The values $K_f = 56$, $K_b = 64$, $K_u = 79$, $K_c = 41$, $K_r = 1$, and $m = 1$ were determined by our tool, while the cipher's structure necessitates that $t = 2$.

4.5 Application to ARADI

ARADI is a block cipher designed with low-latency, introduced by Greene et al. [GMW24] from the US National Security Agency. This section details our findings on ARADI, where our tool attacked 5 rounds of the cipher.

4.5.1 Specification of ARADI

The block and key sizes for ARADI are 128 and 256 bits, respectively. The round function of ARADI is built on the SPN framework and involves three main operations: the S-box layer π, the i-th linear map Λ_i, and the i-th round key addition τ_{k_i}. The full cipher contains 16 rounds and is defined by:

$$\tau_{k_{16}} \circ \bigcirc_{i=0}^{15} (\Lambda_{i \mod 4} \circ \pi \circ \tau_{k_i}), \tag{22}$$

Each of these operations modifies a 128-bit state, structured into four 32-bit words w,x,y,z, representing the state as $w \| x \| y \| z$ or (w,x,y,z).

The S-box Layer π. The S-box layer π modifies the input state (w,x,y,z) as follows:

$$\begin{aligned} x &\leftarrow x \oplus (w \wedge y), \\ z &\leftarrow z \oplus (x \wedge y), \\ y &\leftarrow y \oplus (w \wedge z), \\ w &\leftarrow w \oplus (x \wedge z). \end{aligned} \tag{23}$$

The Linear Layer Λ_i. In the i-th round, the input state (w,x,y,z) is transformed by Λ_i as follows:

$$\Lambda_i(w,x,y,z) \rightarrow (L_i(w),L_i(x),L_i(y),L_i(z)), \tag{24}$$

where L_i is an involutory linear map applied to each 32-bit word. For a 32-bit input to L_i divided into two 16-bit words u and l, the mapping L_i is defined by:

$$(u,l) \rightarrow \Big(u \oplus S_{16}^{a_i}(u) \oplus S_{16}^{c_i}(l), l \oplus S_{16}^{a_i}(l) \oplus S_{16}^{b_i}(u)\Big), \tag{25}$$

where $S_{16}^{m}(\cdot)$ represents a circular left shift of a 16-bit word by m positions. The shift parameters for the linear layer are given in Table 5.

$i \mod 4$	a_i	b_i	c_i
0	11	8	14
1	10	9	11
2	9	4	14
3	8	9	7

Table 5: Shift parameters for the ARADI linear layer.

The Key Addition Layer. This layer `XOR`s a 128-bit round key with the state. Round keys are generated using a separate key schedule algorithm [GMW24]. The ARADI key schedule uses an array of eight 32-bit words K_j and applies two invertible linear maps M_0 and M_1 on pairs of 32-bit words. These mappings are defined as:

$$\begin{aligned} M_0(x,y) &= (S_{32}^1(x) \oplus y, S_{32}^3(y) \oplus S_{32}^1(x) \oplus y) \\ M_1(x,y) &= (S_{32}^9(x) \oplus y, S_{32}^{28}(y) \oplus S_{32}^9(x) \oplus y) \end{aligned} \tag{26}$$

where S_{32} represents a left circular shift on a 32-bit word. The key schedule operates in a two-step cycle. Two steps, beginning with an even-indexed round. At step i, let the register values be $(K_i^0,K_i^1,K_i^2,K_i^3,K_i^4,K_i^5,K_i^6,K_i^7)$. The i-th round key is then $K_i^0 \| K_i^1 \| K_i^2 \| K_i^3$ for even rounds and $K_i^4 \| K_i^5 \| K_i^6 \| K_i^7$ for odd rounds. During each step, four pairs of consecutive words are mixed using either M_0 or M_1. A word permutation P_j is then applied, where $P_0 = (12)(56)$ and $P_1 = (14)(36)$ for j as the round modulo 2. Initially, the eight 32-bit words of the key $K_0,\ldots,K_7$ are loaded, and a counter is `XOR`ed into K_7 at each step to prevent slide/rotational attacks.

4.5.2 Key Recovery Attack on ARADI

In Figure 8, we outline the positions designated as bit addresses in the structure of ARADI.

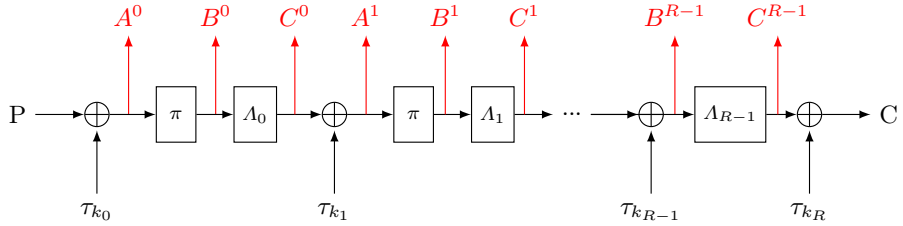

Fig. 8: Attack on ARADI. The positions $A^r,B^r,C^r \quad 0 \le r < R$ are considered as bit addresses.

As shown in the structure of ARADI, the S-box π is applied to each bit of the 4 words (w,x,y,z). Based on this, a look-up table for the S-box π can be produced as follows:

Table 6: The look-up table for the ARADI's S-box π.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	0	1	2	3	4	D	F	6	8	B	5	E	C	7	A	9

Based on the component function of the look-up table in Table 6, the dependency sets for the forward direction are as follows:

$$\begin{aligned} &I_0 = I_2 = \{0,1,2,3\}, \quad J_0 = J_2 = \emptyset \\ &I_1 = \{0,2\}, \quad J_1 = \{1\} \\ &I_3 = \{0,1,2\}, \quad J_3 = \{3\} \end{aligned} \tag{27}$$

For the backward direction, we have:

$$\begin{aligned} &I_0 = \{1,3\}, \quad J_0 = \{0\} \\ &I_1 = I_3 = \{0,1,2,3\}, \quad J_1 = J_3 = \emptyset \\ &I_2 = \{0,1,3\}, \quad J_2 = \{2\} \end{aligned} \tag{28}$$

Representing the set of all key bits as $\mathcal{K} = \{k_0,...,k_{256}\}$, the round keys τ_{k_r} for $0 \leq r \leq R$, derived from $\mathcal{K}$ based on the key schedule and used alternately in the round functions of ARADI, are defined as follows:

$$\begin{aligned} \tau_{k_0} &= \{k_{160},k_{159},...,k_{225}\} \\ \tau_{k_i} &= \{k_{31}+k_{64},k_{30}+k_{63},...,k_{88}+k_{97}+k_{101}\} \\ &\vdots \end{aligned} \tag{29}$$

For ARADI, our tool was able to suggest an attack on 5 rounds with time, data, and memory complexities of $2^{234.8}$, 28, and 2^{19}, respectively, with the matching point being the bits $B^3[22]$. According to the output of our tool, the parameters are $K_f = 171$, $K_b = 230$, $K_u = 249$, $K_c = 152$, $K_r = 7$, and $m = 1$. Given the cipher's structure, the value of t is set to 2.

5 Conclusion and Future Works

In this paper, we presented MITM key recovery attack that requires minimal data and memory resources. Additionally, we proposed an automated tool that optimally measures the complexity of the attack and suggests suitable bits to be used as matching points. We applied our tool to the lightweight block ciphers CRAFT, Midori, WARP, PRESENT80, and ARADI.

Although our proposed MITM technique does not cover the higher number of rounds, it has significant advantages, including low data and memory requirements, and it is not restricted by the type of data used, making it effective with any random data. One notable benefit of our automated tool is its ability to quickly measure the complexity of the attack and suggest matching points, in contrast to other automated techniques that require the code to run for several hours or even days before returning the attack's complexity.

A potential direction for future work is to apply this technique to other ciphers. Additionally, in the proposed technique, the cancellation of key bits that are nonlinearly involved through the linear layers is not considered. Modeling such events could improve the complexity of the attack or extend its coverage to additional rounds.

References

BBI+15. S. Banik, A. Bogdanov, T. Isobe, K. Shibutani, H. Hiwatari, T. Akishita, and F. Regazzoni. Midori: A block cipher for low energy. In *Advances in Cryptology – ASIACRYPT 2015*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015. doi:10.1007/978-3-662-48800-3_17.

BBI+20. S. Banik, Z. Bao, T. Isobe, H. Kubo, F. Liu, K. Minematsu, K. Sakamoto, N. Shibata, and M. Shigeri. WARP: Revisiting GFN for lightweight 128-bit block cipher. In *Selected Areas in Cryptography (SAC)*, volume 12804 of *Lecture Notes in Computer Science*, pages 535–564. Springer, 2020. doi:10.1007/978-3-030-81652-0_21.

BC13. C. Boura and A. Canteaut. A new criterion for avoiding the propagation of linear relations through an Sbox. In *Fast Software Encryption (FSE)*, volume 8424 of *Lecture Notes in Computer Science*, pages 585–604. Springer, 2013. doi:10.1007/978-3-662-43933-3_30.

BDD+23. C. Boura, N. David, P. Derbez, G. Leander, and M. Naya-Plasencia. Differential meet-in-the-middle cryptanalysis. In *Adv. in Cryptology – CRYPTO*, volume 14083 of *Lecture Notes in Computer Science*, pages 240–272. Springer, 2023. doi:10.1007/978-3-031-38548-3_9.

BKL+07. A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems (CHES)*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007. doi:10.1007/978-3-540-74735-2_31.

BLMR19. C. Beierle, G. Leander, A. Moradi, and S. Rasoolzadeh. CRAFT: Lightweight tweakable block cipher with efficient protection against DFA attacks. *IACR Trans. Symmetric Cryptol.*, 2019(1):5–45, 2019. doi:10.13154/tosc.v2019.i1.5-45.

BR10. A. Bogdanov and C. Rechberger. A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher KTANTAN. In *Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2010. doi:10.1007/978-3-642-19574-7_16.

DF16. P. Derbez and P.-A. Fouque. Automatic search of meet-in-the-middle and impossible differential attacks. In *Advances in Cryptology – CRYPTO 2016*, volume 9815 of *Lecture Notes in Computer Science*, pages 157–184. Springer, 2016. doi:10.1007/978-3-662-53008-5_6.

DH77. W. Diffie and M. E. Hellman. Special feature: Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, June 1977. doi:10.1109/c-m.1977.217750.

DS08. H. Demirci and A. A. Selçuk. A meet-in-the-middle attack on 8-round AES. In *Fast Software Encryption (FSE)*, volume 5086 of *Lecture Notes in Computer Science*, pages 116–126. Springer, 2008. doi:10.1007/978-3-540-71039-4_7.

DSP07. O. Dunkelman, G. Sekar, and B. Preneel. Improved meet-in-the-middle attacks on reduced-round DES. In *Progress in Cryptology – INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2007. doi:10.1007/978-3-540-77026-8_8.

GLRW10. J. Guo, S. Ling, C. Rechberger, and H. Wang. Advanced meet-in-the-middle preimage attacks: First results on full Tiger, and improved results on MD4 and SHA-2. In *Advances in Cryptology – ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2010. doi:10.1007/978-3-642-17373-8_4.

GMW24. P. Greene, M. Motley, and B. Weeks. Aradi and llama: Low-latency cryptography for memory encryption. *IACR Cryptol. ePrint Arch.*, 2024.

HE22. H. Hadipour and M. Eichlseder. AutoGuess: A tool for finding guess-and-determine attacks and key bridges. In *Applied Cryptography and Network Security (ACNS)*, volume 13269 of *Lecture Notes in Computer Science*, pages 230–250. Springer, 2022. doi:10.1007/978-3-031-09234-3_12.

IS13. T. Isobe and K. Shibutani. Generic key recovery attack on feistel scheme. In *Advances in Cryptology – ASIACRYPT 2013*, volume 8269 of *Lecture Notes in Computer Science*, pages 464–485. Springer, 2013. doi:10.1007/978-3-642-42033-7_24.

KRS12. D. Khovratovich, C. Rechberger, and A. Savelieva. Bicliques for preimages: Attacks on Skein-512 and the SHA-2 family. In *Fast Software Encryption (FSE)*, volume 7549 of *Lecture Notes in Computer Science*, pages 244–263. Springer, 2012. doi:10.1007/978-3-642-34047-5_15.

LW17. L. Lin and W. Wu. Meet-in-the-middle attacks on reduced-round Midori64. *IACR Trans. Symmetric Cryptol.*, 2017(1):215–239, 2017. doi:10.13154/tosc.v2017.i1.215-239.

MLC23. Z. Ma, M. Li, and S. Chen. Meet-in-the-middle attacks on round-reduced CRAFT based on automatic search. *IET Inf. Secur.*, 17(3):534–543, 2023. doi:10.1049/ise2.12114.

oSN18. National Institute of Standards and Technology (NIST). Lightweight cryptography, 2018.

Sas18. Y. Sasaki. Integer linear programming for three-subset meet-in-the-middle attacks: Application to GIFT. In *Advances in Information and Computer Security (IWSEC)*, volume 11049 of *Lecture Notes in Computer Science*, pages 227–243. Springer, 2018. doi:10.1007/978-3-319-97916-8_15.

SS23. A. Schrottenloher and M. Stevens. Simplified modeling of MITM attacks for block ciphers: New (quantum) attacks. *IACR Trans. Symmetric Cryptol.*, 2023(3):146–183, 2023. doi:10.46586/tosc.v2023.i3.146-183.

ZLW+23. K. Zhang, X. Lai, L. Wang, J. Guan, B. Hu, S. Wang, and T. Shi. Meet-in-the-middle attack with splice-and-cut technique and a general automatic framework. *Des. Codes Cryptogr.*, 91(9):2845–2878, 2023. doi:10.1007/s10623-023-01226-4.