Improved All-Subkeys Recovery Attacks on FOX, KATAN and SHACAL-2 Block Ciphers

Takanori Isobe and Kyoji Shibutani

Sony Corporation
1-7-1 Konan, Minato-ku, Tokyo 108-0075, Japan
{Takanori.Isobe,Kyoji.Shibutani}@jp.sony.com

Abstract. The all-subkeys recovery (ASR) attack is an extension of the meet-in-the-middle attack, which allows evaluating the security of a block cipher without analyzing its key scheduling function. Combining the ASR attack with some advanced techniques such as the function reduction and the repetitive ASR attack, we show the improved ASR attacks on the 7-round reduced FOX64 and FOX128. Moreover, the improved ASR attacks on the 119-, 105- and 99-round reduced KATAN32, KATAN48 and KATAN64, and the 42-round reduced SHACAL-2 are also presented, respectively. As far as we know, all of those attacks are the best single-key attacks with respect to the number of attacked rounds in literature.

Key words: block cipher, meet-in-the-middle attack, all-subkeys recovery attack

1 Introduction

Since the meet-in-the-middle (MITM) attack was applied to KTANTAN [7], a lot of its improvements have been introduced such as the splice-and-cut technique [4], the initial structure [24], the biclique cryptanalysis [6, 19], the internal state guess [14, 10], the sieve-in-the-middle technique [9] and the parallel-cut technique [23]. Since the MITM attack basically exploits the weakness in the key scheduling function, it was believed that a block cipher having a strong key scheduling function has enough immunity against the MITM attack.

Isobe and Shibutani proposed the all-subkeys recovery (ASR) approach at SAC 2012 as an extension of the MITM attack [16], and showed several best attacks on block ciphers having relatively complex key scheduling function including CAST-128 [1], SHACAL-2 [13], FOX [18] and KATAN [8]. One of the advantages of the ASR attack compared to the basic MITM attack is that it does not need to take the key scheduling function into account, since it recovers all subkeys instead of the master key. Thus, it has been shown that the MITM attack may be more applicable to block ciphers. Moreover, the ASR approach enables us to evaluate the lower bounds on the security against key recovery attack for a block cipher structure, since the ASR attack is applicable independently from the underlying key scheduling function. For Feistel schemes, such lower bounds were shown by using the ASR attack with a couple of its improvements such as

Table 1. Summary of Attacks on `FOX64/128`, `KATAN32/48/64` and `SHACAL-2` (Single-Key Setting)

algorithm	attack type	# attacked rounds	time	memory	data	reference
`FOX64`	Integral	5	$2^{109.4}$	Not given	2^9 CP	[26]$*$
	Impossible Diff.	5	2^{80}	Not given	2^{40} CP	[27]$*$
	ASR	**6**	2^{124}	2^{124}	15 CP	this paper
	ASR	**7**	2^{124}	2^{123}	$2^{30.9}$ CP	this paper
`FOX128`	Integral	5	$2^{205.6}$	Not given	2^9 CP	[26]
	Impossible Diff.	5	2^{144}	Not given	2^{72} CP	[27]
	ASR	5	2^{228}	2^{228}	14 KP	[16]
	ASR	**6**	2^{221}	2^{221}	26 CP	this paper
	ASR	**7**	2^{242}	2^{242}	2^{63} CP	this paper
`KATAN32`	ASR	110	2^{77}	$2^{75.1}$	138 KP	[16]
	Differential	114	2^{77}	Not given	$2^{31.9}$ CP	[2]
	ASR	**119**	$2^{79.1}$	$2^{79.1}$	144 CP	this paper
`KATAN48`	ASR	100	2^{78}	2^{78}	128 KP	[16]
	ASR	**105**	$2^{79.1}$	$2^{79.1}$	144 CP	this paper
`KATAN64`	ASR	94	$2^{77.1}$	$2^{79.1}$	116 KP	[16]
	ASR	**99**	$2^{79.1}$	$2^{79.1}$	142 CP	this paper
`SHACAL-2`	ASR	41	2^{500}	2^{492}	244 KP	[16]
	ASR	**42**	2^{508}	2^{508}	2^{25} CP	this paper

$*$ While the 6- and 7-round attacks on `FOX64` were presented in [26, 27], the time complexities of both attacks exceed 2^{128}. Similarly, the optimized exhaustive key-searches on the KATAN family were presented in [28].

the function reduction in [17]. For instance, the function reduction reduces the number of subkeys required to compute the matching state by exploiting degrees of freedom of plaintext/ciphertext pairs. Then, the number of attacked rounds can be increased by the ASR attack. Therefore, in order to more precisely evaluate the security of a block cipher against the ASR attack, the following natural question arises: Are those advanced techniques applicable to other structures such as Lai-Massey and LFSR-type schemes?

In this paper, we first apply the function reduction technique to Lai-Massey, LFSR-type and source-heavy generalized Feistel schemes to extend the ASR attacks on those structures. Then, we further improve the attacks on those structures by exploiting structure dependent properties and optimizing data complexity in the function reduction. For instance, the ASR attack with the function reduction on `FOX` can be improved by using the keyless one-round relation in Lai-Massey scheme. Moreover, combined with the repetitive ASR approach , which optimizes the data complexity when using the function reduction, the attack on `FOX` can be further improved. Those results are summarized in Table 1. As far as we know, all of the results given by this paper are the best single-key attacks with respect to the number of attacked rounds in literature[1] We emphasize that our improvements keep the basic concept of the ASR attack, which enables us to evaluate the security of a block cipher without analyzing its key scheduling function. Therefore, our results are considered as not only the best single-key attacks on the specific block ciphers but also the lower bounds on the security of the target block cipher structures independently from key scheduling functions.

The rest of this paper is organized as follows. Section 2 briefly reviews the previously shown techniques including the all-subkeys recovery approach, the function reduction and the repetitive all-subkeys recovery approach. The improved all-subkeys recovery attacks on `FOX64/128`, `KATAN32/48/64` and `SHACAL-2` are presented in Sections 3, 4 and 5, respectively. Finally, we conclude in Section 6.

2 Preliminary

2.1 All-Subkeys Recovery Approach [16]

The all-subkeys recovery (ASR) attack was proposed in [16] as an extension of the meet-in-the-middle (MITM) attack. Unlike the basic MITM attack, the ASR attack is guessing all-subkeys instead of the master key so that the attack can be constructed independently from the underlying key scheduling function.

Let us briefly review the procedure of the ASR attack. First, an attacker determines an s-bit matching state S in a target n-bit block cipher consisting of R rounds. The state S can be computed from a plaintext P and a set of subkey bits $\mathcal{K}_{(1)}$ by a function $\mathcal{F}_{(1)}$ as $S = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$. Similarly, S can be computed from the corresponding ciphertext C and another set of subkey bits $\mathcal{K}_{(2)}$ by a function $\mathcal{F}_{(2)}$ as $S = \mathcal{F}^{-1}_{(2)}(C, \mathcal{K}_{(2)})$. Let $\mathcal{K}_{(3)}$ be a set of the remaining subkey bits, i.e., $|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}| + |\mathcal{K}_{(3)}| = R \cdot \ell$, where ℓ denotes the size of each subkey. For a plaintext P and the corresponding ciphertext C, the equation $\mathcal{F}_{(1)}(P, \mathcal{K}_{(1)}) = \mathcal{F}^{-1}_{(2)}(C, \mathcal{K}_{(2)})$ holds when the guessed subkey bits $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are correct. Since $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ can be guessed independently, we can efficiently filter out the incorrect subkeys from the key candidates. After this process, it is expected that there will be $2^{R\cdot\ell-s}$ key candidates. Note that the number of key candidates can be reduced by parallel performing the matching with additional plaintext/ciphertext pairs. In fact, using N plaintext/ciphertext pairs, the number of key candidates is reduced to $2^{R\cdot\ell-N\cdot s}$, as long as $N \leq (|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}|)/s$. Finally, the attacker exhaustively searches the correct key from the remaining key candidates. The required computations (i.e. the number of encryption function calls) of the attack in total C_{comp} is estimated as

$$C_{comp} = \max(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N + 2^{R\cdot\ell-N\cdot s}. \tag{1}$$

The number of required plaintext/ciphertext pairs is $\max(N, \lceil (R\cdot\ell - N\cdot s)/n \rceil)$, where n is the block size of the target cipher. The required memory is about

[1] In the related-key setting, the attacks on the 174-, 145-, 130- and 44-round reduced `KATAN32`, `KATAN48`, `KATAN64` and `SHACAL-2` were presented, respectively [20, 15].

$\min(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N$ blocks, which is the cost of the table used for the matching.

2.2 Improvements on All-Subkeys Recovery Approach

In the ASR attack, the number of the subkeys required to compute the state S from P or C, i.e., $\mathcal{K}_{(1)}$ or $\mathcal{K}_{(2)}$, is usually dominant parameter in the required complexities. Thus, in general, reducing those subkeys $\mathcal{K}_1$ and $\mathcal{K}_2$ will make the ASR attack applicable to more rounds. In the followings, we briefly review and introduce a couple of techniques to reduce such subkeys required to compute the matching state.

Function Reduction Technique. For Feistel ciphers, the *function reduction* technique that directly reduces the number of involved subkeys was introduced in [17]. The basic concept of the function reduction is that fixing some plaintext bits, ciphertext bits or both by exploiting degrees of freedom of a plaintext/ciphertext pair allows an attacker to regard a key dependent variable as a new subkey. As a result, substantial subkeys required to compute the matching state are reduced. By using the function reduction, the lower bounds on the security of several Feistel ciphers against generic key recovery attacks were given in [17]. Note that a similar approach was presented in [11] for directly guessing intermediate state values, while in the function reduction, equivalently transformed key values are guessed.

Suppose that the i-th round state S_i is computed from the $(i-1)$-th round state S_{i-1} XORed with the i-th round subkey K_i by the i-th round function G_i, i.e., $S_i = G_i(K_i \oplus S_{i-1})$. For clear understanding, we divide the function reduction into two parts: a key linearization and an equivalent transform as follows.

- **Key Linearization.** Since the i-th round function G_i is a non-linear function, the i-th round subkey K_i cannot pass through G_i by an equivalent transform. The key linearization technique, which is a part of the function reduction, exploits the degree of freedom of plaintexts/ciphertexts to express S_i as a linear relation of S_{i-1} and K_i, i.e., $S_i = L_i(S_{i-1}, K_i)$, where L_i is a linear function. Once S_i is represented by a linear relation of S_{i-1} and K_i, K_i can be forwardly moved to a next non-linear function by an equivalent transform. Note that, if the splice-and-cut technique [4] is used with the key linearization, K_i can be divided into both forward and backward directions.
- **Equivalent Transform.** After the key linearization, the i-th round subkey K_i is replaced with a new subkey K'_i to pass through a non-linear function. However, in order to reduce the involved subkey bits on the trails to the matching state, all-subkeys on the trails affected by K'_i are also replaced with new variables by an equivalent transform. Consequently, the number of subkeys required to compute the matching state can be reduced. For the Feistel ciphers, it is easily done by replacing all-subkeys in the even numbered rounds K_j with $K'_j (= K'_1 \oplus K_j)$, where j is even.

The splice-and-cut technique [4], which was originally presented in the attack of the two-key triple DES [21], was well used in the recent meet-in-the-middle attacks [3, 6, 7, 19, 24]. It regards that the first and last rounds are consecutive by exploiting degree of freedom of plaintext/ciphertexts, and thus any round can be the start point. In general, the splice-and-cut technique is useful to analyze the specific block cipher that key-dependency varies depending on the chunk separation. However, in the ASR approach, the splice-and-cut technique does not work effectively, since the ASR treats all-subkeys as independent variables to evaluate the security independently from the key scheduling function. On the other hand, the function reduction exploits degrees of freedom of plaintexts/ciphertexts to reduce subkey bits required to compute the matching state, and does not use relations among subkeys. Therefore, the function reduction technique is more useful and suitable for the ASR approach than the splice-and-cut technique. However, as mentioned in the description of the key linearization, the combined use of the splice-and-cut and the function reduction in the key linearization is also possible, e.g. the attack on Feistel-1 [17] and the attack on SHACAL-2 in this paper.

Repetitive All-Subkeys Recovery Approach. Since the function reduction exploits the degree of freedom of plaintexts/ciphertexts, it sometimes causes an attack infeasible due to lack of available data. For such cases, we introduce a variant of the all-subkeys recovery approach called *repetitive all-subkeys recovery approach* that repeatedly applies the all-subkeys recovery to detect the correct key. The variant can reduce the required data for each all-subkeys recovery phase, though the total amount of the required data is unchanged. Note that a similar technique, called inner loop technique, was used in [5, 23] for reducing the memory requirements. The repetitive all-subkeys recovery approach is described as follows.

1. Mount the ASR attack with N plaintext/ciphertexts, where N is supposed to be less than $(|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}|)/s$, then put the remaining key candidates into a table T_1. The number of expected candidates is $|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}| - N \cdot s$.
2. Repeatedly mount the ASR attack with *different* N plaintext/ciphertexts. If the remaining candidate match with ones in T_1, such candidates are put into another table T_2. The number of expected candidates is $|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}| - 2 \cdot N \cdot s$.
3. Repeat the above processes until the correct key is found, i.e., $M = (|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}|)/(N \cdot s)$ times.

When the above procedure is repeated M (≥ 2) times, the computational costs to detect $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are estimated as

$$C_{comp} = (\max(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N) \times M + (2^{|\mathcal{K}_{(1)}|+|\mathcal{K}_{(2)}|-N \cdot s}) + \cdots + (2^{|\mathcal{K}_{(1)}|+|\mathcal{K}_{(2)}|-(M-1) \cdot N \cdot s}).$$

While the required data in total is $(|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}|)/s$ $(= ((|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}|)/(M \cdot s)) \cdot M)$, each ASR approach is done with $N = (|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}|)/(M \cdot s)$ data,

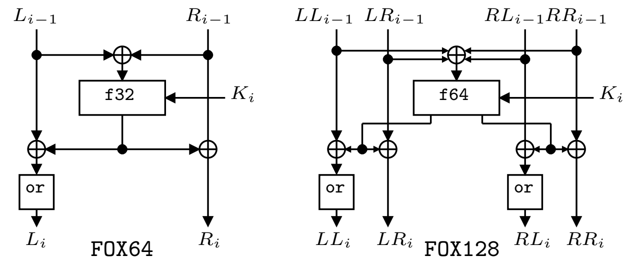


Fig. 1. Round Functions of FOX64 and FOX128

which is M times less than that required in the basic ASR attack. The required memory is about $\max(2^{|\mathcal{K}_{(1)}|+|\mathcal{K}_{(2)}|-N\cdot s}, \min(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N)$ blocks, which is the cost for the table used in the matching. We demonstrate the effectiveness of the proposed variant in the attack on the reduced FOX in Section 3.

3 Improved All-Subkeys Recovery Attacks on FOX64 and FOX128

In this section, we present the improved ASR attacks using the function reduction and the repetitive ASR approach on the 6- and 7-round reduced FOX64 and FOX128 block ciphers. After short descriptions of FOX64 and FOX128, the function reduction on FOX64 is presented. Then, we show how to construct the attack on the 6-round FOX64, and how to extend it to the 7-round variant by using the repetitive ASR approach. Similarly, the function reduction on FOX128, the attack on the 6-round FOX128, and the attack on the 7-round FOX128 with the repetitive ASR approach are introduced, respectively.

3.1 Descriptions of FOX64 and FOX128

FOX [18], also known as IDEA-NXT, is a family of block ciphers designed by Junod and Vaudenay in 2004. FOX employs a Lai-Massey scheme including two variants referred as FOX64 and FOX128 (see Fig. 1).

FOX64 is a 64-bit block cipher consisting of a 16-round Lai-Massey scheme with a 128-bit key. The i-th round 64-bit input state is denoted as two 32-bit words $(L_{i-1} \mid\mid R_{i-1})$. The i-th round function updates the input state using the 64-bit i-th round key K_i as follows:

$$(L_i||R_i) = (\texttt{or}(L_{i-1} \oplus \texttt{f32}(L_{i-1} \oplus R_{i-1}, K_i))||R_{i-1} \oplus \texttt{f32}(L_{i-1} \oplus R_{i-1}, K_i)),$$

where $\texttt{or}(x_0||x_1) = (x_1||(x_0 \oplus x_1))$ for 16-bit x_0, x_1. f32 outputs a 32-bit data from a 32-bit input X and two 32-bit subkeys LK_i and RK_i as $(\texttt{sigma4}(\texttt{mu4}(\texttt{sigma4}(X \oplus LK_i)) \oplus RK_i) \oplus LK_i)$, where sigma4 denotes the S-box layer consisting of four 8-bit S-boxes and mu4 denotes the 4×4 MDS matrix. Two 32-bit subkeys LK_i and RK_i are derived from K_i as $K_i = (LK_i||RK_i)$.

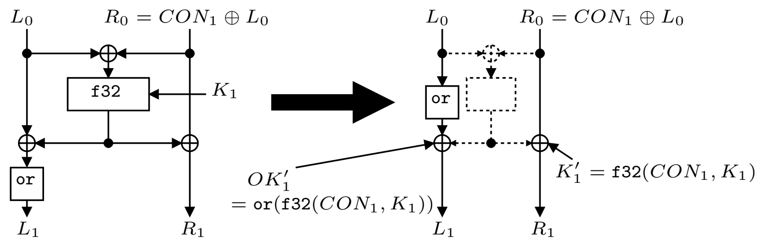


Fig. 2. Key Linearization of `FOX64`

`FOX128` is a 128-bit block cipher consisting of a 16-round modified Lai-Massey scheme with a 256-bit key. The i-th round 128-bit input state is denoted as four 32-bit words $(LL_{i-1} \mid\mid LR_{i-1} \mid\mid RL_{i-1} \mid\mid RR_{i-1})$. The i-th round function updates the input state using the 128-bit i-th round key K_i as follows:

$$(LL_i||LR_i) = (\texttt{or}(LL_{i-1} \oplus \phi_L)||LR_{i-1} \oplus \phi_L),$$
$$(RL_i||RR_i) = (\texttt{or}(RL_{i-1} \oplus \phi_R)||RR_{i-1} \oplus \phi_R),$$

where $(\phi_L||\phi_R) = \texttt{f64}((LL_{i-1} \oplus LR_{i-1})||(RL_{i-1} \oplus RR_{i-1}), K_i)$. `f64` outputs a 64-bit data from a 64-bit input X and two 64-bit subkeys LK_i and RK_i as $(\texttt{sigma8}(\texttt{mu8}(\texttt{sigma8}(X \oplus LK_i)) \oplus RK_i) \oplus LK_i)$, where `sigma8` denotes the S-box layer consisting of eight 8-bit S-boxes and `mu8` denotes the 8×8 MDS matrix. Two 64-bit subkeys LK_i and RK_i are derived from K_i as $K_i = (LK_i||RK_i)$.

3.2 Function Reduction on `FOX64`

Key Linearization (Fig. 2). If the value of $L_0 \oplus R_0$ is fixed to a constant CON_1, the input of `f32` is fixed as $\texttt{f32}(CON_1, K_1)$. By regarding $\texttt{f32}(CON_1, K_1)$ as a 32-bit new key K'_1, K'_1 is XORed to L_0 and R_0. Since `or` is a linear operation, the state after the first round is expressed as $(L_1||R_1) = (\texttt{or}(L_0) \oplus OK'_1)||(R_0 \oplus K'_1)$, where $OK'_1 = \texttt{or}(K'_1)$ (see Fig. 2). This implies that the first round keys linearly affect L_1 and R_1.

Equivalent Transform (Fig. 3). In the second round, OK'_1 and K'_1 are XORed with LK_2 in the first and last operations of `f32` function. Let $LK'_2 = LK_2 \oplus K'_1 \oplus OK'_1$, $K1'' = K1' \oplus LK2$, and $OK1'' = \texttt{or}(OK'1 \oplus LK2)$ be new keys. Then `f32` function contains K'_2 $(= LK'_2||RL_2)$, and K''_1 and $OK1''$ linearly affect outputs of the second round.

In the third round, OK''_1 and K''_1 are also XORed with LK_3 in the first and last operations of `f32` function. Let $LK'_3 = LK_3 \oplus K''_1 \oplus OK''_1$, $K1''' = K1'' \oplus LK2$, and $OK1''' = \texttt{or}(OK''1 \oplus LK2)$ be new keys (see Fig. 3).

Note that the same technique can be applied to the inverse of `FOX64`, because the round function of `FOX64` has the involution property.

3.3 Attack on the 6-Round FOX64

In this attack, we use the following one-round keyless linear relation of the Lai-Massey construction.

$$\mathtt{or}^{-1}(L_{i+1}) \oplus R_{i+1} = L_i \oplus R_i.$$

From this equation, the 16-bit relation is obtained as follows

$$((L_4^{(1)} \oplus L_4^{(3)})||L_4^{(3)}) \oplus (R_4^{(3)}||R_4^{(1)}) = (L_3^{(3)}||L_3^{(1)}) \oplus (R_3^{(3)}||R_3^{(1)}),$$

where $L_i^{(j)}$ and $R_i^{(j)}$ are the j-th byte of L_i and R_i, respectively, and $L_i^{(3)}$ and $R_i^{(3)}$ are the most significant bytes ,i.e., $L_i = \{L_i^{(3)}||L_i^{(2)}||L_i^{(1)}||L_i^{(0)}\}$ and $R_i = \{R_i^{(3)}||R_i^{(2)}||R_i^{(1)}||R_i^{(0)}\}$.

Forward Computation in $\mathcal{F}_{(1)}$: For given $\{K_2'$, LK_3', $RK_3'^{(3)}$, $RK_3'^{(1)}$, $K_1'''^{(3)}$, $K_1'''^{(1)}$, $OK_1'''^{(3)}$, $OK_1'''^{(1)}\}$, $(L_3^{(3)}||L_3^{(1)}) \oplus (R_3^{(3)}||R_3^{(1)})$ is computable. Since $(K_1'''^{(3)}$ $||$ $K_1'''^{(1)})$ and $(OK_1'''^{(3)}||OK_1'''^{(1)})$ linearly affect $(L_3^{(3)}||L_3^{(1)})$ and $(R_3^{(3)}||R_3^{(1)})$, respectively, we can regard $(K_1'''^{(3)}||K_1'''^{(1)}) \oplus (OK_1'''^{(3)}||OK_1'''^{(1)})$ as a new 16-bit key $XORK_1$. Then, $(L_3^{(3)}||L_3^{(1)}) \oplus (R_3^{(3)}||R_3^{(1)})$ is obtained from $112(= 64 + 32 + 8 + 8)$ bits of the key $\{K_2', LK_3', RK_3'^{(3)}, RK_3'^{(1)}\}$ and linearly-dependent 16-bit key $XORK_1$.

Backward Computation in $\mathcal{F}_{(2)}$: $((L_4^{(1)} \oplus L_4^{(3)})||L_4^{(3)}) \oplus (R_4^{(3)}||R_4^{(1)})$ is obtained from 112 (=64 + 32 + 16) bits of the key $\{K_6$, LK_5, $RK_5^{(1)}$, $RK_5^{(3)}\}$. Using the indirect matching technique [3], 8 bits out of 16 bits of $XORK_1$ are moved to the left half of the matching equation. Then, the left and right halves of the equation contains 120 bits of the key, i.e., $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 120$.

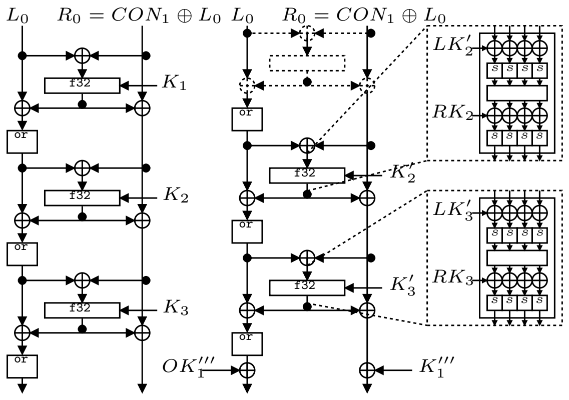
Evaluation. When the parameter $N = 15$, the time complexity for finding the involved 240-bit key is estimated as

$$C_{comp} = \max(2^{120}, 2^{120}) \times 15 + 2^{240-15\cdot16} = 2^{124}.$$

The required data for the attack is only 15 (=$\max(15, \lceil(240-15\cdot16)/64\rceil)$) chosen plaintext/ciphertext pairs, and the required memory is estimated as about 2^{124} (=$\min(2^{120}, 2^{120}) \times 15$) blocks.

3.4 Attack on the 7-Round FOX64

If the function reduction is applied as well in the backward direction, the 7-round attack is feasible, i.e., the relation of $L_7 \oplus R_7$ is fixed to a constant CON_2. Due to the involution property of the FOX64 round function, $((L_4^{(1)} \oplus L_4^{(3)})$ $||$ $L_4^{(3)})$



$$
\begin{aligned}
K'_1 &= \mathtt{f32}(CON_1, K_1)\\
OK'_1 &= \mathtt{or}(K'_1)\\
LK'_2 &= LK_2 \oplus K'_1 \oplus OK'_1\\
K''_1 &= K'_1 \oplus LK_2\\
OK''_1 &= \mathtt{or}(OK'_1 \oplus LK_2)\\
LK'_3 &= LK_3 \oplus K''_1 \oplus OK''_1\\
K'''_1 &= K''_1 \oplus LK_3\\
OK'''_1 &= \mathtt{or}(OK''_1 \oplus LK_3)\\
K'_2 &= LK'_2 || RK_2\\
K'_3 &= LK'_3 || RK_3
\end{aligned}
$$

Fig. 3. Function Reduction of FOX64

$\oplus$ $(R_4^{(3)} || R_4^{(1)})$ is also obtained from 112 (=64 + 32 + 8 + 8) bits of the key and linearly-dependent 16-bit key $XORK_2$. In this attack, we further regard $XORK_1 \oplus XORK_2$ as a 16-bit new key. Then, similar to the attack on the 6-round FOX64, the left and right halves of the equation contain 120 bits of the key, i.e., $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 120$.

Repetitive ASR Approach. Recall that plaintexts and ciphertexts need to satisfy the 32-bit relations, $L_0 \oplus R_0 = CON_1$ and $L_7 \oplus R_7 = CON_2$. The required data for finding such pairs is equivalently estimated as the game that an attacker finds 32-bit multicollisions by 32-bit-restricted inputs. It has been known that an n-bit t-multicollision is found in $t!^{1/t} \cdot 2^{n \cdot (t-1)/t}$ random data with high probability [25].

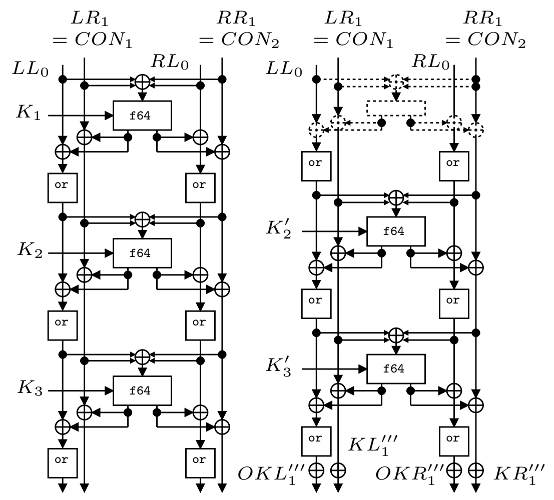
In the basic ASR approach, at least 15 (= 240/16) multicollisions are necessary to detect the 240-bit involved key. To obtain such pairs with a high probability, it require $2^{32.55} (= 15!^{1/15} \cdot 2^{32 \cdot (14)/15})$ plaintext/ciphertext pairs. However, it is infeasible, since the degree of freedom of plaintexts is only 32 bits.

In order to overcome this problem, we utilize the repetitive all-subkeys recovery approach with $M = 2$ variant. In each all-subkeys recovery phase, the required data is reduced to 8 and 7. Then, such eight 32-bit multicollisions are obtained from $2^{29.9}$ plaintext/ciphertext pairs with a high probability. Thus, we can obtain the required data by exploiting free 32 bits.

Evaluation. The time complexity for finding the involved 240 bits key is estimated as

$$C_{comp} = (\max(2^{120}, 2^{120}) \times 8) \times 2 + (2^{240 - 8 \cdot 16}) = 2^{124}.$$

The remaining $208 (= 448 - 240)$ bits are obtained by recursively applying all-subkeys recovery attacks. The time complexity for this phase is roughly estimated as $2^{106 (= 208/2 + 2)}$ using 4 $(= \lceil 208/64 \rceil)$ plaintext/ciphertext pairs.



$KL'_1||KR'_1 = \texttt{f64}(CON_1||CON_2, K_1)$
$OKL'_1 = \texttt{or}(KL'_1)$
$OKR'_1 = \texttt{or}(KR'_1)$

$LK'_2 = LK_2 \oplus$
$((OKL'_1 \oplus KL'_1)||(OKR'_1 \oplus KR'_1))$
$LK_2 = LKL_2||LKR_2$
$KL''_1 = KL'_1 \oplus LKL_2$
$KR''_1 = KR'_1 \oplus LKR_2$
$OKL''_1 = \texttt{or}(OKL'_1 \oplus LKL_2)$
$OKR''_1 = \texttt{or}(OKR'_1 \oplus LKR_2)$
$K'_2 = LK'_2||RK_2$

$LK'_3 = LK_3 \oplus$
$((OKL''_1 \oplus KL''_1)||(OKR''_1 \oplus KR''_1))$
$LK_3 = LKL_3||LKR_3$
$KL'''_1 = KL''_1 \oplus LKL_3$
$KR'''_1 = KR''_1 \oplus LKR_3$
$OKL'''_1 = \texttt{or}(OKL''_1 \oplus LKL_3)$
$OKR'''_1 = \texttt{or}(OKR''_1 \oplus LKR_3$
$K'_3 = LK'_3||RK_3$
$RK_3 = RKL_3||RKR_3$

Fig. 4. Function Reduction of `FOX128`

The required data is $2^{30.9}(= 2^{29.9} \times 2)$ plaintext/ciphertext pairs, and the required memory is about 2^{123} (=max$(2^{240-128}, \min(2^{120}, 2^{120}) \times 8)$) blocks.

3.5 Function Reduction on `FOX128`

Key Linearization (Fig. 4). If two 16-bit relations of $LL_0 \oplus LR_0$ and $RL_0 \oplus RR_0$ are fixed to CON_1 and CON_2, respectively, the input of `f64` is fixed as $\texttt{f64}(CON_1 \mid\mid CON_2, K_1)$. By regarding $\texttt{f64}(CON_1||CON_2, K_1)$ as a 64-bit new key $K'_1 = KL'_1||KR'_1$, KL'_1 and KR'_1 are XORed to $\{LR_0$ and $LR_0\}$ and $\{RR_0$ and $RR_0\}$, respectively. The state after the first round is expressed as follows (see Fig. 4).

$$(LL_1||LR_1||RL_1||RR_1) =$$
$$(\texttt{or}(LL_0) \oplus OKL'_1)||(LR_0 \oplus KL'_1)||(\texttt{or}(RL_0) \oplus OKR'_1)||(RR_0 \oplus KR'_1),$$

where $OKL'_1 = \texttt{or}(KL'_1)$ and $OKR'_1 = \texttt{or}(KR'_1)$. This implies that the first round keys linearly affect LL_1, LR_1, RL_1 and RR_1.

Equivalent Transform (Fig. 4). The equivalent transform is done similar to `FOX64` as shown in Fig. 4.

3.6 Attack on the 6-Round `FOX128`

We use the following one-round keyless linear relation of the modified Lai-Massey construction,

$$\texttt{or}^{-1}(LL_{i+1}) \oplus LR_{i+1} = LL_i \oplus LR_i.$$

From this equation, the 16-bit relation is obtained as follows:

$$((LL_4^{(1)} \oplus LL_4^{(3)})||LL_4^{(3)}) \oplus (LR_4^{(3)}||LR_4^{(1)}) = (LL_3^{(3)}||LL_3^{(1)}) \oplus (LR_3^{(3)}||LR_3^{(1)}).$$

Forward Computation in $\mathcal{F}_{(1)}$: For given $\{K'_2,\ LK'_3,\ RKL_3'^{(3)},\ RKL_3'^{(1)},\ KL_1'''^{(3)},\ KL_1'''^{(1)},\ OKL_1'''^{(3)},\ OKL_1'''^{(1)}\}$, $(LL_3^{(3)}||LL_3^{(1)}) \oplus (LR_3^{(3)}||LR_3^{(1)})$ is computable. Since $(KL_1'''^{(3)}||KL_1'''^{(1)})$ and $(OKL_1'''^{(3)}||OKL_1'''^{(1)})$ linearly affect the matching states $(LL_3^{(3)}||LL_3^{(1)})$ and $(LR_3^{(3)}||LR_3^{(1)})$, respectively, we are able to regard $(LK_1'''^{(3)}||LK_1'''^{(1)}) \oplus (OKL_1'''^{(3)}||OKL_1'''^{(1)})$ as a new 16-bit key $XORK_1$. Then, $(LL_3^{(3)}||LL_3^{(1)}) \oplus (LR_3^{(3)}||LR_3^{(1)})$ is obtained from 208 (=128 + 64 + 8 + 8) bits of the key $\{K'_2, LK'_3, RKL_3'^{(3)}, RKL_3'^{(1)}\}$ and linearly-dependent 16 bits key $XORK_1$.

Backward Computation in $\mathcal{F}_{(2)}$: $((LL_4^{(1)} \oplus LL_4^{(3)})||LL_4^{(3)}) \oplus (LR_4^{(3)}||LR_4^{(1)})$ is obtained from 208 (=128 + 64 + 16) bits of the key $\{K_6,\ LK_5,\ RKL_5^{(1)},\ RKL_5^{(3)}\}$. Using the indirect matching technique, 8 bits out of 16-bit $XORK_1$ are moved to the left half of the matching equation. Then, the left and right halves of the equation contain 216 bits of the key, i.e., $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 216$.

Evaluation. When the parameter $N = 26$, the time complexity for the involved 432 bits is estimated as

$$C_{comp} = \max(2^{216}, 2^{216}) \times 26 = 2^{221}.$$

The remaining 352 (= 768 − 416) bits are obtained by recursively applying the all-subkeys recovery attack. The time complexity for determining the remaining subkeys is roughly estimated as $2^{177.6(=352/2+1.6)}$ using 2 (= $\lceil 352/128 \rceil$) plaintext/ciphertext pairs.

The required data is only 26 chosen plaintext/ciphertext pairs, and the required memory is about 2^{221} (=$\min(2^{216}, 2^{216}) \times 26$) blocks.

3.7 Attack on the 7-Round FOX128

If the function reduction is also used in the backward direction, the 7-round attack is feasible, i.e., two 16-bit relations of $LL_7 \oplus LR_7$ and $RL_7 \oplus RR_7$ are fixed to CON_3 and CON_4, respectively.

Due to the involution property of the FOX128 round function, $((LL_4^{(1)} \oplus LL_4^{(3)})\ ||\ LL_4^{(3)}) \oplus (LR_4^{(3)}||LR_4^{(1)})$ is also obtained from 208 (=128 + 64 + 8 + 8) bits of the key and linearly-dependent 16 bits key $XORK_2$. In this attack, we further regard $XORK_1 \oplus XORK_2$ as a 16 bit new key. Then, similar to the attack on the 6-round FOX128, the left and right halves of the equation contain 216 bits of the key, i.e., $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 216$.

Repetitive ASR Approach. Recall that plaintexts and ciphertexts need to satisfy 64-bit (32×2) relations, $LL_0 \oplus LR_0$ and $RL_0 \oplus RR_0$, and $LL_7 \oplus LR_7$ and $RL_7 \oplus RR_7$, respectively. The cost is equivalently estimated as the game that an attacker finds 64-bit multicollisions with 64-bit-restricted inputs.

In the basic ASR approach, at least 27(= 432/16) multicollisions are needed to detect the 432-bit involved key. To obtain such pairs with a high probability, it requires $2^{65.1}(= 27!^{1/27} \cdot 2^{64 \cdot (26)/27})$ pairs. However, it is infeasible, since the degree of freedom of plaintexts is only 64 bits.

We utilize the repetitive all-subkeys recovery approach with $M = 2$ variant. In each all-subkeys recovery phase, the required data is reduced to 13 and 14. Such 14 64-bit multicollisions are obtained, given $2^{62.0}$ plaintext/ciphertext pairs with high probability.

Evaluation. The time complexity for finding involved 432 bits of the key is estimated as

$$C_{comp} = (\max(2^{216}, 2^{216}) \times 14) \times 2 + 2^{432-16\cdot 14} = 2^{224}.$$

The remaining 480(= 896−432) bits are obtained by recursively applying the all-subkeys recovery attack. The time complexity for this phase is roughly estimated as $2^{242(=480/2+2)}$ using 4 ($= \lceil 480/128 \rceil$) plaintext/ciphertext pairs.

The required data is $2^{63.0}(= 2^{62.0} \times 2)$ plaintext/ciphertext pairs, and the required memory is about 2^{242} blocks.

4 Improved All-Subkeys Recovery Attacks on KATAN32/48/64

In this section, we show that the function reduction techniques are applicable to KATAN32/48/64, then we improve the ASR attacks on KATAN32/48/64 block ciphers by 9, 5 and 5 rounds, respectively.

After a short description of KATAN, we show how to apply the function reduction to KATAN32 in detail. Then, the detailed explanation for the attack on the 119-round reduced KATAN32 is given. For KATAN48 and KATAN64, the detailed explanations for applying the function reductions are omitted, since the analysis is done similar to KATAN32.

4.1 Description of KATAN

KATAN [8] family is a feedback shift register-based block cipher consisting of three variants: KATAN32, KATAN48 and KATAN64 whose block sizes are 32-, 48- and 64-bit, respectively. All of the KATAN ciphers use the same key schedule accepting an 80-bit key and 254 rounds. The plaintext is loaded into two shift registers L_1 and L_2. In each round, L_1 and L_2 are shifted by one bit, and the least significant bits of L_1 and L_2 are updated by $f_b(L_2)$ and $f_a(L_1)$, respectively. The bit functions f_a and f_b are defined as follows:

$$f_a(L_1) = L_1[x_1] \oplus L_1[x_2] \oplus (L_1[x_3] \cdot L_1[x_4]) \oplus (L_1[x_5] \cdot IR) \oplus k_{2i},$$
$$f_b(L_2) = L_2[y_1] \oplus L_2[y_2] \oplus (L_2[y_3] \cdot L_2[y_4]) \oplus (L_2[y_5] \cdot L_2[y_6]) \oplus k_{2i+1},$$

Table 2. Parameters of KATAN Family

Algorithm	$\lvert L_1\rvert$	$\lvert L_2\rvert$	x_1	x_2	x_3	x_4	x_5	y_1	y_2	y_3	y_4	y_5	y_6
KATAN32	13	19	12	7	8	5	3	18	7	12	10	8	3
KATAN48	19	29	18	12	15	7	6	28	19	21	13	15	6
KATAN64	25	39	24	15	20	11	9	38	25	33	21	14	9

where $L[x]$ denotes the x-th bit of L, IR denotes the round constant, and k_{2i} and k_{2i+1} denote the 2-bit i-th round key. Note that for KATAN family, the round number starts from 0 instead of 1, i.e., KATAN family consists of round functions starting from the 0-th round to the 253-th round. L_1^i or L_2^i denote the i-th round registers L_1 or L_2, respectively. Let IR^i be the i-th round constant. For KATAN48 or KATAN64, in each round, the above procedure is iterated twice or three times, respectively. All of the parameters for the KATAN ciphers are listed in Table 2.

The key scheduling function of KATAN ciphers copies the 80-bit user-provided key to $k_0, ..., k_{79}$, where $k_i \in \{0,1\}$. Then, the remaining 428 bits of the round keys are generated as follows:

$$k_i = k_{i-80} \oplus k_{i-61} \oplus k_{i-50} \oplus k_{i-13} \qquad \text{for } i = 80, ..., 507.$$

4.2 Function Reduction on KATAN32

Key Linearization. In the i-th round function of KATAN32, two key bits k_{2i} and k_{2i+1} are linearly inserted into states $L_1^i[0]$ and $L_2^i[0]$, respectively, these states are not updated in the i-th round. Thus, the key linearization technique is not necessary.

Equivalent Transform. Let us consider how linearly-inserted key bits are used in the following round functions. For instance, k_1 is linearly inserted to $L_1^1[0]$, and the updated state $L_1^1[0]$ is described as $(X[0] \oplus k_1)$, where $X[i]$ is defined as

$$X[i] = L_2^0[18-i] \oplus L_2^0[7-i] \oplus (L_2^0[12-i] \cdot L_2^0[10-i]) \oplus (L_2^0[8-i] \cdot L_2^0[3-i]),$$

where $L_2^0[-i] = L_2^i[0]$. For computing $f_a(L_1)$, the state $L_1^1[0] = (X[0] \oplus k_1)$ is used in the following five positions,

- $L_2^4[3]$: $((X[0] \oplus k_1) \cdot IR^4)$ is XORed with k_8. If $X[0]$ is fixed to a constant CON_1, a new key k'_8 is defined as $((CON_1 \oplus k_1) \cdot IR^4) \oplus k_8$.
- $L_2^6[5]$: $((X[0] \oplus k_1) \cdot L_1^6[8]) = ((X[0] \oplus k_1) \cdot L_1^0[2])$ is XORed with k_{12}. If $L_1^0[2]$ is also fixed to a constant CON_2, a new key k'_{12} is defined as $((CON_1 \oplus k_1) \cdot CON_2) \oplus k_{12}$.
- $L_2^8[7]$: $(X[0] \oplus k_1)$ is directly XORed with k_{16}. A new key k'_{16} is defined as $(CON_1 \oplus k_1) \oplus k_{16}$.

Table 3. Conditions for 8-bit Function Reductions

Key bit	State bits to be fixed
k_1	$X[0]$, $L_1^0[2]$, $X[3]$
k_3	$X[1]$, $L_1^0[1]$, $X[4]$
k_5	$X[2]$, $L_1^0[0]$, $X[5]$
k_7	$X[3]$, $X[0]$, $X[6]$
k_9	$X[4]$, $X[1]$, $X[7]$
k_{11}	$X[5]$, $X[2]$, $X[8]$
k_{13}	$X[6]$, $X[3]$, $X[9]$
k_{15}	$X[7]$, $X[4]$, $X[10]$

– $L_2^9[8] : ((X[0] \oplus k_1) \cdot L_1^9[5]) = ((X[0] \oplus k_1) \cdot (X[3] \oplus k_7))$ is XORed with k_{18}. If $X[3]$ is also fixed to a constant CON_3, a new key k'_{18} is defined as $((CON_1 \oplus k_1) \cdot (CON_3 \oplus k_7)) \oplus k_{18}$.
– $L_2^{13}[12] : (X[0] \oplus k_1)$ is directly XORed with k_{26}. A new key k'_{26} is defined as $(CON_1 \oplus k_1) \oplus k_{26}$.

Thus, by fixing $X[0]$, $L_1^0[2]$ and $X[3]$ to constants and defining new key bits k'_8, k'_{12}, k'_{16}, k'_{18} and k'_{26}, we can omit one key bit k_1, i.e., we can compute without k_1 in the forward direction. Note that CON_1, CON_2 and CON_3 are not restricted to constant values. Even if CON_1, CON_2 and CON_3 are expressed by only key bits, we can define new keys in the same manner.

Conditions for Function Reduction. Table 3 shows conditions for 8-bit function reductions. If these 14 bits of $L_1^0[0]$, $L_1^0[2]$, $L_1^0[2]$, $X[0], \dots, X[10]$ are fixed to constants or expressed by only key bits, then we can eliminate 8 bits of the key, k_1, k_3, k_5, k_7, k_9, k_{11}, k_{13} and k_{15}, in the forward computation of `KATAN32`.

Let us explain how to control $X[0]$ and $X[10]$ by exploiting the degree of freedom of plaintexts. $X[0]$ to $X[10]$ are expressed as follows:

$$\begin{aligned}
X[0] &= \underline{L_2^0[18]} \oplus L_2^0[7] \oplus (L_2^0[12] \cdot L_2^0[10]) \oplus (L_2^0[8] \cdot L_2^0[3]),\\
X[1] &= \underline{L_2^0[17]} \oplus L_2^0[6] \oplus (L_2^0[11] \cdot L_2^0[9]) \oplus (L_2^0[7] \cdot L_2^0[2]),\\
X[2] &= \underline{L_2^0[16]} \oplus L_2^0[5] \oplus (L_2^0[10] \cdot L_2^0[8]) \oplus (L_2^0[6] \cdot L_2^0[1]),\\
X[3] &= \underline{L_2^0[15]} \oplus L_2^0[4] \oplus (L_2^0[9] \cdot L_2^0[7]) \oplus (L_2^0[5] \cdot L_2^0[0]),\\
X[4] &= \underline{L_2^0[14]} \oplus L_2^0[3] \oplus (L_2^0[8] \cdot L_2^0[6]) \oplus (\underline{L_2^0[4]} \cdot (Y[0] \oplus k_0)),\\
X[5] &= \underline{L_2^0[13]} \oplus L_2^0[2] \oplus (L_2^0[7] \cdot L_2^0[5]) \oplus (\underline{L_2^0[3]} \cdot (Y[1] \oplus k_2)),\\
X[6] &= \underline{L_2^0[12]} \oplus L_2^0[1] \oplus (L_2^0[6] \cdot L_2^0[4]) \oplus (\underline{L_2^0[2]} \cdot (Y[2] \oplus k_4)),\\
X[7] &= \underline{L_2^0[11]} \oplus L_2^0[0] \oplus (L_2^0[5] \cdot L_2^0[3]) \oplus (\underline{L_2^0[1]} \cdot (Y[3] \oplus k_6)),\\
X[8] &= \underline{L_2^0[10]} \oplus (Y[0] \oplus k_0) \oplus (L_2^0[4] \cdot L_2^0[2]) \oplus (\underline{L_2^0[0]} \cdot (Y[4] \oplus k_8)),\\
X[9] &= \underline{L_2^0[9]} \oplus (Y[1] \oplus k_2) \oplus (L_2^0[3] \cdot L_2^0[1]) \oplus ((\underline{Y[0]} \oplus k_0) \cdot (\underline{Y[5]} \oplus k_{10}),\\
X[10] &= \underline{L_2^0[8]} \oplus (Y[2] \oplus k_4) \oplus (L_2^0[2] \cdot L_2^0[0]) \oplus ((\underline{Y[1]} \oplus k_2) \cdot (\underline{Y[6]} \oplus k_{12}),
\end{aligned}$$

where

$$\begin{aligned}
Y[0] &= \underline{L_1^0[12]} \oplus L_1^0[7] \oplus (L_1^0[5] \cdot L_1^0[8]) \oplus (L_1^0[3] \cdot IR^0),\\
Y[1] &= \underline{L_1^0[11]} \oplus L_1^0[6] \oplus (L_1^0[4] \cdot L_1^0[7]) \oplus (L_1^0[2] \cdot IR^1),\\
Y[2] &= L_1^0[10] \oplus L_1^0[5] \oplus (L_1^0[3] \cdot L_1^0[6]) \oplus (L_1^0[1] \cdot IR^2),\\
Y[3] &= L_1^0[9] \oplus L_1^0[4] \oplus (L_1^0[2] \cdot L_1^0[5]) \oplus (L_1^0[0] \cdot IR^3),\\
Y[4] &= L_1^0[8] \oplus L_1^0[3] \oplus (L_1^0[1] \cdot L_1^0[4]) \oplus (X[0] \cdot IR^4),\\
Y[5] &= \underline{L_1^0[7]} \oplus L_1^0[2] \oplus (L_1^0[0] \cdot L_1^0[3]) \oplus (X[1] \cdot IR^5),\\
Y[6] &= \underline{L_1^0[6]} \oplus L_1^0[1] \oplus (X[0] \cdot L_1[2]) \oplus (X[2] \cdot IR^6).
\end{aligned}$$

$X[0], \ldots, X[3]$ are easily fixed to constants by controlling 4 bits of $L_2^0[18]$, $L_2^0[17]$, $L_2^0[16]$ and $L_2^0[15]$ (4-bit condition). $X[4], \ldots, X[8]$ contain key bits in AND operations. If $L_2^0[4] = L_2^0[3] = L_2^0[2] = L_2^0[1] = L_2^0[0] = 0$, those key bits are omitted from the equations (5-bit condition). Then $X[4]$ to $X[7]$ are also fixed to constants by controlling 4 bits of $L_2^0[14]$, $L_2^0[13]$, $L_2^0[12]$ and $L_2^0[11]$ (4 bit condition). In $X[8]$, k_0 is also linearly inserted. If $(L_2^0[10] \oplus Y[0] \oplus (L_2^0[4] \cdot L_2^0[2]))$ is fixed to a constant by controlling $L_2^0[10]$, then $X[8]$ is expressed as the form of $CON \oplus k_0$, which depends only on key bits, where CON is an arbitrary constant.

In $X[9]$ and $X[10]$, 4 bits of $Y[0]$, $Y[1]$, $Y[5]$ and $Y[6]$ are needed to be fixed. These values are controlled by $L_1^0[12]$, $L_1^0[11]$, $L_1^0[7]$ and $L_1^0[6]$. If the other bits are fixed by $L_2^0[9]$ and $L_2^0[8]$, $X[9]$ and $X[10]$ are expressed by only key bits.

Therefore, if plaintexts satisfy 23 $(= 3 + 4 + 5 + 4 + 1 + 4 + 2)$ bit conditions described in Table 3, 8 bits of the key are able to be omitted when mounting the ASR attack.

Procedure for Creating Plaintexts. We show how to create plaintexts satisfying the conditions. By using the equations of $X[0]$ to $X[10]$ and $Y[0]$ to $Y[6]$, such plaintexts are easily obtained as follows.

1. Set 18 predetermined values of $L_1^0[0]$, $L_1^0[1]$, $L_1^0[2]$, $X[0], \ldots, X[10]$, $Y[0]$, $Y[1]$, $Y[5]$ and $Y[6]$.
2. Choose values of free 9 bits of $L_2^0[5]$, $L_2^0[6]$, $L_2^0[7]$, $L_1^0[3]$, $L_1^0[4]$, $L_1^0[5]$, $L_1^0[8]$, $L_1^0[9]$ and $L_1^0[10]$.
3. Obtain $L_2^0[8], \ldots, L_2^0[13]$ from equations of $X[5], \ldots, X[10]$, and $L_1^0[6]$ and $L_1^0[7]$ from equations of $Y[5]$ and $Y[6]$, respectively.
4. Obtain $L_2^0[14], \ldots, L_2^0[18]$ from equations of $X[0], \ldots, X[4]$, and $L_1^0[11]$ and $L_1^0[12]$ from equations of $Y[0]$ and $Y[1]$, respectively.
5. Repeat steps 2 to 4 until the required number of plaintexts are obtained.

4.3 Attacks on 119-Round `KATAN32`

Let us consider the 119-round variant of KATAN32 starting from the first (0-th) round. In this attack, $L_2^{69}[18]$ is chosen as the matching state.

Table 4. Conditions for 4-bit Function Reductions

Key bit	State bits to be fixed
k_1	$X'[0]$, $X'[1]$, $L_1^0[6]$, $L_1^0[7]$, $X'[8]$, $X'[9]$
k_3	$X'[2]$, $X'[3]$, $L_1^0[4]$, $L_1^0[5]$, $X'[10]$, $X'[11]$
k_5	$X'[4]$, $X'[5]$, $L_1^0[2]$, $L_1^0[3]$, $X'[12]$, $X'[13]$
k_7	$X'[6]$, $X'[7]$, $L_1^0[0]$, $L_1^0[1]$, $X'[14]$, $X'[15]$

Forward Computation in $\mathcal{F}_{(1)}$: $L_2^{69}[18]$ depends on 83 subkey bits. This implies that $L_2^{69}[18]$ can be computed by a plaintext P and 83 bits of subkeys. More specifically, $L_2^{69}[18] = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{k_0, ..., k_{70}, k_{72}, \ldots, k_{76}, k_{80}, k_{83}, k_{84}, k_{85}, k_{89}, k_{93}, k_{100}\}$ and $|\mathcal{K}_{(1)}| = 83$. If the function reduction technique with the 23-bit condition of plaintexts is used, 8 bits of $\{k_1, k_3, k_5, k_7, k_9, k_{11}, k_{13}, k_{15}\}$ can be omitted in computations of $\mathcal{F}_{(1)}$. Thus, $L_2^{69}[18]$ is computable with $75(= 83-8)$ bits. In addition, since 4 bits of $\{k_{68}, k_{75}, k_{85}, k_{100}\}$ linearly affect $L_2^{69}[18]$, we can regard $k_{68} \oplus k_{75} \oplus k_{85} \oplus k_{100}$ as a new key k_f. Thus, $72(= 75 - 4 + 1)$ bits are involved in the forward computation.

Backward Computation in $\mathcal{F}_{(2)}$: In the backward computation starting from the 118-th round, the matching state $L_2^{69}[18]$ is computed as $L_2^{69}[18] = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{k_{138}, k_{150}, k_{154}, k_{158}, k_{160}, k_{162}, k_{165}, k_{166}, k_{168}, k_{170}, k_{172}, \ldots k_{237}\}$, and $|\mathcal{K}_{(2)}| = 76$. Since 4 bits of $\{k_{138}, k_{160}, k_{165}, k_{175}\}$ linearly affect $L_2^{69}[18]$, we can regard $k_{138} \oplus k_{160} \oplus k_{165} \oplus k_{175}$ as a new key k_b. Furthermore, by the indirect matching, k_b is moved to the forward computation, then $k_b \oplus k_f$ is regarded as a new key in $\mathcal{F}_{(1)}$. Thus, $72(= 76 - 4)$ bits are involved in the backward computation.

Evaluation. For the 119-round reduced KATAN32, the matching state S is chosen as $L_2^{69}[18]$ (1-bit state). When $N = 144$ $(\leq (72+72)/1)$, the time complexity for finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ is estimated as

$$C_{comp} = \max(2^{72}, 2^{72}) \times 144 = 2^{79.1}.$$

The required data is only 144 chosen plaintext/ciphertext pairs in which the 23 bits of each plaintext satisfy conditions. The required memory is about $2^{79.1}$ blocks.

Finally, we need to find the remaining $94(= 119 \times 2 - 144)$ bits of subkeys by using the simple MITM approach in the setting where $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are known. The time complexity and the required memory for this process are roughly estimated as $2^{49}(= 2^{48} + 2^{46})$ and 2^{46} blocks, respectively. These costs are obviously much less than those of finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$.

4.4 Function Reduction on KATAN48

Table 4 shows conditions for 4-bit function reductions, where $X'[i]$ is defined as:

$$X'[i] = L_2^0[28-i] \oplus L_2^0[19-i] \oplus (L_2^0[21-i] \cdot L_2^0[13-i]) \oplus (L_2^0[15-i] \cdot L_2^0[6-i]).$$

If these values are fixed to target constants, we can eliminate the key bits in the computation of KATAN48.

$X'[0], \ldots, X'[6]$ are fixed by controlling 7 bits of $L_2^0[22], \ldots, L_2^0[28]$ (7 bit condition). $X'[7], \ldots, X'[15]$ contain key bits in AND operations. If $L_2^0[8] = L_2^0[7] =, \ldots, = L_2^0[1] = L_2^0[0] = 0$, these key bits are omitted from these equations (9 bit condition). Then $X'[7], \ldots, X'[15]$ are also fixed by controlling 9 bits of $L_2^0[13], \ldots, L_2^0[21]$ (9 bit condition).

Therefore, if plaintexts satisfy 33 $(= 8 + 7 + 9 + 9)$ bit conditions described in Table 4, 4 bits of the key are able to be omitted when mounting the ASR attack.

4.5 Attacks on 105-Round KATAN48

Let us consider the 105-round variant of KATAN48 starting from the first (0-th) round. In this attack, $L_2^{61}[28]$ is chosen as the matching state.

Forward Computation in $\mathcal{F}_{(1)}$: $L_2^{61}[28]$ depends on 79 subkey bits. This implies that $L_2^{61}[28]$ can be computed by a plaintext P and 79 bits of subkeys. More specifically, $L_2^{61}[28] = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{k_0, ..., k_{68}, k_{70}, k_{71}, k_{72}, k_{75}, k_{77}, k_{78}, k_{81}, k_{85}, k_{87}, k_{92}\}$ and $|\mathcal{K}_{(1)}| = 79$. If the function reduction technique with the 33-bit condition of plaintexts is used, 4 bits of k_1, k_3, k_5, k_7 can be omitted in computations of $\mathcal{F}_{(1)}$. Thus, $L_2^{61}[28]$ is computable with $75 (= 79 - 4)$ bits. In addition, 3 bits of $\{k_{75}, k_{81}, k_{92}\}$ linearly affect $L_2^{61}[28]$. Thus, we can regard $k_{75} \oplus k_{81} \oplus k_{92}$ as a new key. By using indirect matching, $k_f = k_{75} \oplus k_{81} \oplus k_{92}$ is moved to $\mathcal{F}_{(2)}$. Then, $72 (= 75 - 3)$ bits are involved in the forward computation.

Backward Computation in $\mathcal{F}_{(2)}$: In the backward computation starting from the 104-th round, the matching state $L_2^{61}[28]$ is computed as $L_2^{61}[28] = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{k_{122}, k_{128}, k_{130}, k_{134}, k_{136}, k_{138}, k_{140}, k_{141}, k_{142}, k_{144}, \ldots k_{209}\}$, and $|\mathcal{K}_{(2)}| = 75$. 4 bits of $\{k_{122}, k_{130}, k_{140}, k_{141}\}$ linearly affect $L_2^{61}[28]$. Thus, we can regard $k_b = k_{122} \oplus k_{130} \oplus k_{140} \oplus k_{141}$ as a new key. Furthermore, we define $k_f \oplus k_b$ as a new key. Then, $72 (= 75 - 4 + 1)$ bits are involved in the backward computation.

Evaluation. For the 105-round reduced KATAN48, the matching state S is chosen as $L_2^{61}[28]$ (1-bit state). When $N = 144$ $(\leq (72 + 72)/1)$, the time complexity for finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ is estimated as

$$C_{comp} = \max(2^{72}, 2^{72}) \times 144 = 2^{79.1}.$$

The required data is only 144 chosen plaintext/ciphertext pairs. The required memory is about $2^{79.1}$ blocks.

Finally, we need to find the remaining 66 $(= 105 \times 2 - 144)$ bits of subkeys by using the simple MITM approach in the setting where $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are known.

Table 5. Conditions for 2-bit Function Reductions

Key bit	State bits to be fixed
k_1	$X''[0]$, $X''[1]$, $X''[2]$, $L_1^0[6]$, $L_1^0[7]$, $L_1^0[8]$, $X''[9]$, $X''[10]$, $X''[11]$,
k_3	$X''[3]$, $X''[4]$, $X''[5]$, $L_1^0[3]$, $L_1^0[4]$, $L_1^0[5]$, $X''[12]$, $X''[13]$, $X''[14]$,

The time complexity and the required memory for this process are roughly estimated as $2^{34}(= 2^{34} + 2^{32})$ and 2^{32} blocks, respectively. These costs are obviously much less than those of finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$.

4.6 Function Reduction on `KATAN64`

Table 5 shows conditions for 2-bit function reductions, where $X''[i]$ is defined as:

$$X''[i] = L_2^0[38-i] \oplus L_2^0[25-i] \oplus (L_2^0[33-i] \cdot L_2^0[21-i]) \oplus (L_2^0[14-i] \cdot L_2^0[9-i]).$$

If these values are fixed to target constants, we can eliminate the key bits in the computation of `KATAN64`.

$X''[0], \ldots, X''[9]$ are fixed by controlling 10 bits of $L_2^0[29], \ldots, L_2^0[38]$ (10 bit condition). $X''[10]$, ..., $X''[14]$ contain key bits in AND operations. If $L_2^0[4] = , \ldots, = L_2^0[0] = 0$, these key bits are omitted from these equations (5 bit condition). Then $X''[10], \ldots, X''[14]$ are also fixed by controlling 5 bits $L_2^0[18], ..., L_2^0[23]$ (5 bit condition).

Therefore, if plaintexts satisfy 29 (= 9 + 10 + 5 + 5) bit conditions described in Table 5, 2 bits of the key are able to be omitted when mounting the ASR attack.

4.7 Attacks on 99-Round `KATAN64`

Let us consider the 99-round variant of `KATAN64` starting from the first (0-th) round. In this attack, $L_2^{57}[38]$ is chosen as the matching state.

Forward Computation in $\mathcal{F}_{(1)}$: $L_2^{57}[38]$ depends on 74 subkey bits. This implies that $L_2^{57}[38]$ can be computed by a plaintext P and 74 bits of subkeys. More specifically, $L_2^{57}[38] = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{k_0, ..., k_{66}, k_{70}, k_{71}, k_{72}, k_{75}, k_{77}, k_{81}, k_{88}\}$ and $|\mathcal{K}_{(1)}| = 74$. If the function reduction technique with the 29-bit condition of plaintexts is used, 2 bits of k_1, k_3 can be omitted in computations of $\mathcal{F}_{(1)}$. Thus, $L_2^{57}[38]$ is computable with $72(= 74 - 2)$ bits. In addition, 3 bits of $\{k_{71}, k_{77}, k_{88}\}$ linearly affect $L_2^{57}[38]$. Thus, we can regard $k_{71} \oplus k_{77} \oplus k_{88}$ as a new key. Then, $70(= 72 - 3 + 1)$ bits are involved in the forward computation.

Backward Computation in $\mathcal{F}_{(2)}$: In the backward computation starting from the 98-th round, the matching state $L_2^{57}[38]$ is computed as $L_2^{57}[38] = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{k_{114}, k_{116}, k_{120}, k_{122}, k_{124}, k_{126}, k_{128}, k_{130}, \ldots k_{197}\}$, and $|\mathcal{K}_{(2)}| = 75$. 3 bits of $\{k_{114}, k_{122}, k_{131}\}$ linearly affect $L_2^{57}[38]$. Thus, we can consider $k_{114} \oplus k_{122} \oplus k_{131}$ as a new key, and move it to the forward computation by the indirect matching. Then, $72(= 75 - 3)$ bits are involved in the backward computation.

Evaluation. For the 99-round reduced `KATAN64`, the matching state S is chosen as $L_2^{57}[38]$ (1-bit state).

When $N = 142$ $(\leq (72 + 70)/1)$, the time complexity for finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ is estimated as

$$C_{comp} = \max(2^{72}, 2^{70}) \times 142 = 2^{79.1}.$$

The required data is only 142 chosen plaintext/ciphertext pairs. The required memory is about $2^{77.1}$ blocks.

Finally, we need to find the remaining $56(= 99 \times 2 - 142)$ bits of subkeys by using the simple MITM approach in the setting where $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are known. The time complexity and the required memory for this process are roughly estimated as 2^{28} and 2^{28} blocks, respectively. These costs are obviously much less than those of finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$.

5 Improved All-Subkeys Recovery Attack on `SHACAL-2`

This section presents the ASR attacks on `SHACAL-2` with the function reduction techniques. Then, we propose a 42-round attack on `SHACAL-2`, based on the 41-round attack on `SHACAL-2` [13].

5.1 Description of SHACAL-2

SHACAL-2 [13] is a 256-bit block cipher based on the compression function of SHA-256 [12]. It was submitted to the NESSIE project and selected in the NESSIE portfolio [22].

SHACAL-2 inputs the plaintext to the compression function as the chaining variable, and inputs the key to the compression function as the message block. First, a 256-bit plaintext is divided into eight 32-bit words A_0, B_0, C_0, D_0, E_0, F_0, G_0 and H_0. Then, the state update function updates eight 32-bit variables, A_i, B_i, ..., G_i, H_i in 64 steps as follows:

$$\begin{aligned}
T_1 &= H_i \boxplus \Sigma_1(E_i) \boxplus Ch(E_i, F_i, G_i) \boxplus K_i \boxplus W_i,\\
T_2 &= \Sigma_0(A_i) \boxplus Maj(A_i, B_i, C_i),\\
A_{i+1} &= T_1 \boxplus T_2,\ B_{i+1} = A_i,\ C_{i+1} = B_i,\ D_{i+1} = C_i,\\
E_{i+1} &= D_i \boxplus T_1,\ F_{i+1} = E_i,\ G_{i+1} = F_i,\ H_{i+1} = G_i,
\end{aligned}$$

where K_i is the i-th step constant, W_i is the i-th step key (32-bit), and the functions Ch, Maj, Σ_0 and Σ_1 are given as follows:

$$\begin{aligned}
Ch(X,Y,Z) &= XY \oplus \overline{X}Z,\\
Maj(X,Y,Z) &= XY \oplus YZ \oplus XZ,\\
\Sigma_0(X) &= (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22),\\
\Sigma_1(X) &= (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25).
\end{aligned}$$

After 64 steps, the function outputs eight 32-bit words A_{64}, B_{64}, C_{64}, D_{64}, E_{64}, F_{64}, G_{64} and H_{64} as the 256-bit ciphertext. Hereafter p_i denotes the i-th step state, i.e., $p_i = A_i||B_i||...||H_i$.

The key scheduling function of SHACAL-2 takes a variable length key up to 512 bits as the inputs, then outputs 64 32-bit step keys. First, the 512-bit input key is copied to 16 32-bit words W_0, W_1, ..., W_{15}. If the size of the input key is shorter than 512 bits, the key is padded with zeros. Then, the key scheduling function generates 48 32-bit step keys $(W_{16}, ..., W_{63})$ from the 512-bit key $(W_0, ..., W_{15})$ as follows:

$$W_i = \sigma_1(W_{i-2}) \boxplus W_{i-7} \boxplus \sigma_0(W_{i-15}) \boxplus W_{i-16}, (16 \leq i < 64),$$

where the functions $\sigma_0(X)$ and $\sigma_1(X)$ are defined by

$$\begin{aligned}
\sigma_0(X) &= (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3),\\
\sigma_1(X) &= (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10).
\end{aligned}$$

5.2 Function Reduction on `SHACAL-2`

In the round function of `SHACAL-2`, a round key W_i is inserted to the state T_i by an arithmetic addition operation. We show that the splice and cut framework is applicable by using the partial key linearization technique.

The computation of T_1 is expressed as

$$T_1 = (H_i \boxplus W_i) \boxplus \Sigma_1(E_i) \boxplus Ch(E_i, F_i, G_i) \boxplus K_i.$$

In a straight way, the computation of $(H_i \boxplus W_i)$ is not divided into two parts as $(HL_i \boxplus WL_i)||(HR_i \boxplus WR_i)$ due to the carry bit between these computations, where HL_i and WL_i denote the higher x-bits of H_i and W_i, respectively, and HR_i and WR_i are the lower $(32-x)$-bits of H_i and W_i. If HR_i is fixed to 0, it is equivalent to $(HL_i \boxplus WL_i)||(HL_i \oplus WR_i)$. Then, it allows us to independently compute these two parts without dealing with carry bits. Therefore, by using the splice and cut framework, 32 key bits of one round is divided into forward and backward computations as shown in Fig. 5.

However, we can not reduce the number of involved key bits by using an equivalent transform. It is because that the involved 32-bit key W_i is used at least eight times in the forward and backward directions. In order to fully control values in each state, more than $512(32 \times 8)$ bits of conditions are required.

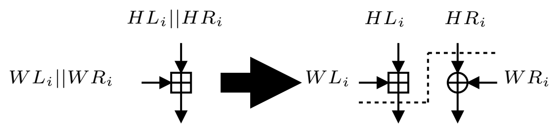


Fig. 5. Splice and Cut of SHACAL-2

5.3 Attacks on 42-Round SHACAL-2

We show that the splice and cut framework [4] is applicable to SHACAL-2 by using the key linearization technique. Then we extend the 41-round attack [16] by one more round. In particular, the splice and cut technique is done in the first round, and the higher 15 bits are move to the backward computation, and the lower 17 bits are move to the forward computation. Then we choose the lowest 1 bit of A_{17} as the matching point.

Forward Computation in $\mathcal{F}_{(1)}$: The lowest 1 bit of A_{17} can be computed from the 16-th state p_{16} and the lowest 1 bit of W_{16}, since the other bits of W_{16} are not affected to the lower 1 bit of A_{17}. Thus, the matching state S (the lowest 1 bit of A_{17}) is calculated as $S = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in$ {the lower 17 bits of $W_0, W_1, ..., W_{15}$, the lowest 1 bit of W_{16}} and $|\mathcal{K}_{(1)}| = 498 (= 32 \times 15 + 1 + 17)$.

Backward Computation in $\mathcal{F}_{(2)}$: We utilize the following observation [16].

Observation 1 *The lower t bits of A_{j-10} are obtained from the j-th state p_j and the lower t bits of three subkeys W_{j-1}, W_{j-2} and W_{j-3}.*

From Observation 1, the matching state S (the lowest 1 bit of A_{17}) can be computed as $S = \mathcal{F}^{-1}_{(2)}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in$ {the higher 15 bits of W_0, $W_{27}, ..., W_{41}$, the lowest 1 bits of W_{24}, W_{25} and W_{26}}. Thus, $|\mathcal{K}_{(2)}| = 498 (= 32 \times 15 + 1 \times 3 + 15)$.

Evaluation. The matching state S is the lowest 1 bit of A_{17}, $|\mathcal{K}_{(1)}| = 498$ and $|\mathcal{K}_{(2)}| = 498$. Thus, using 996 chosen plaintext/ciphertext pairs (i.e. $N = 244 \leq (498 + 498)/1$), the time complexity for finding all-subkeys is estimated as

$$C_{comp} = \max(2^{498}, 2^{498}) \times 996 + 2^{1344-996} = 2^{508}.$$

The required data is $2^{25} (= 996 \times 2^{15})$ chosen plaintext/ciphertext pairs, since 15 bits of plaintext are not controlled in the backward computation when using the splice and cut technique. The required memory is 2^{508} $(= \min(2^{498}, 2^{498}) \times 996)$ blocks.

6 Conclusion

The concept of the ASR attack is quite simple, which recovers all-subkeys instead of the master key, but useful to evaluate the security of block cipher structures without analyzing key scheduling functions. Thus, it is valuable to study its improvements to design a secure block cipher. We first observed the function reduction technique, which improved the ASR attack and was originally applied to Feistel schemes. Then, with some improvements such as the repetitive ASR approach, we applied the function reduction to other block cipher structures including Lai-Massey, generalized Lai-Massey, LFSR-type and source-heavy generalized Feistel schemes.

As applications of our approach, we presented the improved ASR attacks on the 7-, 7-, 119-, 105-, 99-, and 42-round reduced `FOX64`, `FOX128`, `KATAN32`, `KATAN48`, `KATAN64` and `SHACAL-2`. All of our results updated the number of attacked rounds by the previously known best attacks. We emphasize that our attacks work independently from the structure of the key scheduling function. In other words, strengthening the key scheduling function does not improve the security against our attack. It implies that our results give the lower bounds on the security of the target structures such as Lai-Massey scheme rather than the specific block ciphers against generic key recovery attack. Therefore, we believe that our results are useful for a deeper understanding the security of the block cipher structures.

References

1. C. Adams, "The CAST-128 encryption algorithm." RFC-2144, May 1997.
2. M. R. Albrecht and G. Leander, "An all-in-one approach to differential cryptanalysis for small block ciphers." in *SAC* (L. R. Knudsen and H. Wu, eds.), vol. 7707 of *LNCS*, pp. 1–15, Springer, 2013.
3. K. Aoki, J. Guo, K. Matusiewicz, Y. Sasaki, and L. Wang, "Preimages for step-reduced SHA-2." in *ASIACRYPT* (M. Matsui, ed.), vol. 5912 of *LNCS*, pp. 578–597, Springer, 2009.
4. K. Aoki and Y. Sasaki, "Preimage attacks on one-block MD4, 63-step MD5 and more." in *SAC* (R. Avanzi, L. Keliher, and F. Sica, eds.), vol. 5381 of *LNCS*, pp. 103–119, Springer, 2008.
5. E. Biham, O. Dunkelman, N. Keller, and A. Shamir, "New data-efficient attacks on reduced-round IDEA." *IACR Cryptology ePrint Archive*, vol. 2011, p. 417, 2011.
6. A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique cryptanalysis of the full AES." in *ASIACRYPT* (D. H. Lee and X. Wang, eds.), vol. 7073 of *LNCS*, pp. 344–371, Springer, 2011.
7. A. Bogdanov and C. Rechberger, "A 3-subset meet-in-the-middle attack: Cryptanalysis of the lightweight block cipher KTANTAN." in *SAC* (A. Biryukov, G. Gong, and D. R. Stinson, eds.), vol. 6544 of *LNCS*, pp. 229–240, Springer, 2010.
8. C. D. Cannière, O. Dunkelman, and M. Knežević, "KATAN and KTANTAN - a family of small and efficient hardware-oriented block ciphers." in *CHES* (C. Clavier and K. Gaj, eds.), vol. 5747 of *LNCS*, pp. 272–288, Springer, 2009.

9. A. Canteaut, M. Naya-Plasencia, and B. Vayssière, “Sieve-in-the-middle: Improved MITM attacks.” in *CRYPTO (1)* (R. Canetti and J. A. Garay, eds.), vol. 8042 of *LNCS*, pp. 222–240, Springer, 2013.
10. I. Dinur, O. Dunkelman, and A. Shamir, “Improved attacks on full GOST.” in *FSE* (A. Canteaut, ed.), vol. 7549 of *LNCS*, pp. 9–28, Springer, 2012.
11. O. Dunkelman, G. Sekar, and B. Preneel, “Improved meet-in-the-middle attacks on reduced-round DES.” in *INDOCRYPT* (K. Srinathan, C. P. Rangan, and M. Yung, eds.), vol. 4859 of *LNCS*, pp. 86–100, Springer, 2007.
12. FIPS, “Secure Hash Standard (SHS).” Federal Information Processing Standards Publication 180-4.
13. H. Handschuh and D. Naccache, “SHACAL.” NESSIE Proposal (updated), Oct. 2001. Available from `https://www.cosic.esat.kuleuven.be/nessie/updatedPhase2Specs/SHACAL/shacal-tweak.zip`.
14. T. Isobe, “A single-key attack on the full GOST block cipher.” in *FSE* (A. Joux, ed.), vol. 6733 of *LNCS*, pp. 290–305, Springer, 2011.
15. T. Isobe, Y. Sasaki, and J. Chen, “Related-key boomerang attacks on KATAN32/48/64.” in *ACISP* (C. Boyd and L. Simpson, eds.), vol. 7959 of *LNCS*, pp. 268–285, Springer, 2013.
16. T. Isobe and K. Shibutani, “All subkeys recovery attack on block ciphers: Extending meet-in-the-middle approach.” in *SAC* (L. R. Knudsen and H. Wu, eds.), vol. 7707 of *LNCS*, pp. 202–221, Springer, 2013.
17. T. Isobe and K. Shibutani, “Generic key recovery attack on Feistel scheme.” in *ASIACRYPT (1)* (K. Sako and P. Sarkar, eds.), vol. 8269 of *LNCS*, pp. 464–485, Springer, 2013.
18. P. Junod and S. Vaudenay, “FOX : A new family of block ciphers.” in *SAC* (H. Handschuh and M. A. Hasan, eds.), vol. 3357 of *LNCS*, pp. 114–129, Springer, 2004.
19. D. Khovratovich, G. Leurent, and C. Rechberger, “Narrow-bicliques: Cryptanalysis of full IDEA.” in *EUROCRYPT* (D. Pointcheval and T. Johansson, eds.), vol. 7237 of *LNCS*, pp. 392–410, Springer, 2012.
20. J. Lu and J. Kim, “Attacking 44 rounds of the SHACAL-2 block cipher using related-key rectangle cryptanalysis.” *IEICE Transactions*, vol. 91-A, no. 9, pp. 2588–2596, 2008.
21. R. C. Merkle and M. E. Hellman, “On the security of multiple encryption.” *Commun. ACM*, vol. 24, no. 7, pp. 465–467, 1981.
22. NESSIE consortium, “NESSIE portfolio of recommended cryptographic primitives.” 2003. Available from `https://www.cosic.esat.kuleuven.be/nessie/deliverables/decision-final.pdf`.
23. I. Nikolic, L. Wang, and S. Wu, “The parallel-cut meet-in-the-middle attack.” *IACR Cryptology ePrint Archive*, vol. 2013, p. 530, 2013.
24. Y. Sasaki and K. Aoki, “Finding preimages in full MD5 faster than exhaustive search.” in *EUROCRYPT* (A. Joux, ed.), vol. 5479 of *LNCS*, pp. 134–152, Springer, 2009.
25. K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota, “Birthday paradox for multi-collisions.” in *ICISC* (M. S. Rhee and B. Lee, eds.), vol. 4296 of *LNCS*, pp. 29–40, Springer, 2006.
26. W. Wu, W. Zhang, and D. Feng, “Integral cryptanalysis of reduced FOX block cipher.” in *ICISC* (D. Won and S. Kim, eds.), vol. 3935 of *LNCS*, pp. 229–241, Springer, 2005.

27. Z. Wu, Y. Luo, X. Lai, and B. Zhu, "Improved cryptanalysis of the FOX block cipher." in *INTRUST* (L. Chen and M. Yung, eds.), vol. 6163 of *LNCS*, pp. 236–249, Springer, 2009.
28. B. Zhu and G. Gong, "Guess-then-meet-in-the-middle attacks on the KTANTAN family of block ciphers." *IACR Cryptology ePrint Archive*, vol. 2011, p. 619, 2011.