Chosen-Key Distinguishers on 12-Round Feistel-SP and 11-Round Collision Attacks on Its Hashing Modes(Full version)

Xiaoyang Dong[1] and Xiaoyun Wang[1,2]⋆

[1] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, China
`dongxiaoyang@mail.sdu.edu.cn`
[2] Institute for Advanced Study, Tsinghua University
`xiaoyunwang@tsinghua.edu.cn`

Abstract. Since Knudsen and Rijmen proposed the *known-key* attacks in ASIACRYPT 2007, the *open-key* model becomes more and more popular. As the other component of the *open-key* model, *chosen-key* model was applied to the full attacks on AES-256 by Biryukov *et al.* in CRYPTO 2009. In this paper, we explore how practically does the *chosen-key* model affect the real-world cryptography and show that 11-round generic Feistel-SP block cipher is no longer safe in its hashing mode (MMO and MP mode) as there exists collision attacks. This work improves Sasaki and Yasuda's collision attacks by 2 rounds with two interesting techniques. First, we for the first time leverage the available degrees of freedom in the key to reduce the complexity of the inbound phase, which extends the previous 5-round inbound differential to a 7-round one. This results in a 12-round *chosen-key* distinguisher of Feistel-SP block cipher. Second, inspired by the idea of Wang *et al.*, we construct collisions using two blocks. The *rebound attack* is used in the second compression function. We carefully tradeoff between the freedom of the first block and the complexity of the *rebound attack*, and extends the *chosen-key* attack to a 11-round collision attack on its hashing modes (MMO and MP mode).
Keywords: Block cipher, Feistel-SP, Chosen-key, Rebound attack, Hash mode.

1 Introduction

Nowadays, both block ciphers and hash functions are important primitives in cryptography. In many cases, hash functions are based on block ciphers. For instance, if a block cipher and a hash function are both needed in a resource-restricted environment, such as smart cards, RFID tag, nodes in cars or other

⋆ Corresponding author

machines which works in very tiny embedded systems, many applications utilize a block cipher to construct a hash function in order to minimize the design and implementation cost. There are many popular schemes to construct hash functions based on a block cipher, including the Davies-Meyer(DM), Matyas-Meyer-Oseas(MMO) and Miyaguchi-Preneel(MP) hashing modes, which are all included in the PGV hashing schemes [21]. So to evaluate the security of block ciphers used in these schemes is very important.

Different from the classical block cipher security analysis, which relies on the fact that the key value is kept secret, the key value is known to the attackers in these hashing schemes. Recently, Knudsen and Rijmen [15] have proposed to consider the *known-key* attacks on AES. In their attacks, the key is known and the goal is to find two input messages that satisfy some relations. While, in some cases the key is under the control of the attackers, for instance when the key plays a role of salt that is added to the hash functions. This attack model is called *chosen-key* model which has been evaluated and popularized by Biryukov *et al.* in [4]. Both models belong to the *open-key* model.

Feistel block cipher adopts an efficient Feistel network design [9], which is widely trusted and has a long history in cryptography. Nowadays, many block cipher standards such as DES [5], Triple-DES, MISTY1, Camellia and CAST-128 [12] are based on Feistel design. In order to analyze the Feistel primitives comprehensively, Isobe and Shibutani [14] classify them into three types, called $Feistel$-1/2/3. In this paper, we focus on the generic $Feistel$-3 type, which has round functions based on substitution-permutation network(SPN), i.e. the round function starts with an XOR of a subkey, followed by a layer of S-Boxes and a linear diffusion layer. We denote $Feistel$-3 as Feistel-SP block cipher in this paper. In ASIACRYPT 2014, Guo *et al.*[10] gave a 10-round meet-in-the-middle key-recovery attack on generic Feistel-SP block cipher. When the Feistel-SP block cipher is used to construct hash functions, the analysis of resistance of collision attack is immediately needed.

Related Work.

Knudsen and Rijmen in [15] have been the firsts to consider *known-key* distinguishers on AES and Feistel schemes. Besides, they present a half-collision when applying the *known-key* attack to MMO-hashing function with 7-round Feistel block cipher whose round function consists of a round-key XOR followed by an arbitrary key-independent transformmation[3]. The main motivations for this model are summarized by paper [8] as following:

1. If there is no distinguisher when the key is known, then there will also be no distinguisher when the key is secret;

[3] Actually, it is *Feistel*-2 type [14], and can be trivially extend to the Feistel-SP block cipher.

2. If it is possible to find an efficient distinguisher, finding partial collision on the output of the cipher more efficiently than birthday paradox would predict even though the key is known, then the authors would not recommend the use of such cipher;
3. Finally, such model where the key is known or chosen can be interesting to study the use of cipher in a compression function for a hash function.

In [4], Biryukov *et al.* studied the *chosen-key* distinguisher for the full 256-bit key AES. They showed that in time $q \cdot 2^{67}$, it is possible to construct q-differential multicollisions on Davis-Meyer compression function using AES-256, whereas for an ideal cipher, it would require $q \cdot 2^{\frac{q-1}{q+1}128}$. Then the *chosen-key* distinguisher is translated into a key-recovery attack on the full AES-256 in related-key setting. Lamberger *et al.* [17] presented a *chosen-key* distinguisher on the full Whirlpool compression function by taking advantage of rebound techniques and the available degrees of freedom of the key. In [20], Nikolić *et al.* studied the *known-key* and *chosen-key* distinguishers on Feistel and Substitution-Permutation Networks(SPN).

At FSE 2011, by using rebound attack, Sasaki and Yasuda [24] found a 5-round inbound differential path of the Feistel-SP block cipher, by extending 3 rounds forward and backward, an 11-round *known-key* distinguisher was given. Then, they successfully extended the theoretical *known-key* distinguisher to real-world threat by introducing 9-round collisions to its hashing mode (MMO and MP mode). In [23], Sasaki *et al.* studied the *known-key* scenario for Feistel ciphers like Camellia. Later, Sasaki [22] studied the 4-branch generalised Feistel networks with double SP-functions in *known-key* setting, and he left an open problem in this paper that if *chosen-key* scenario could be applied to the study of the Feistel schemes.

Our Contributions.

In this work, we continue to explore how the *open-key* model shake the real-world cryptography. We give an answer to Sasaki's open problem and show that *chosen-key* scenario works better in the study of the generic Feistel-SP block cipher and its hashing mode. We extend Sasaki and Yasuda's 5-round inbound differential path to a 7-round one. Then a 12-round *chosen-key* distinguisher is presented. By exploring the Feistel-SP block cipher used in MMO/MP-hashing mode, a 11-round full-collision attack is constructed which improves Sasaki and Yasuda's collision attacks by 2 rounds. This result shows that the 11-round generic Feistel-SP block cipher is not secure in its hashing mode (MMO and MP mode). It should be noted that our 11-round collision attack is in the same setting as Sasaki and Yasuda's 9-round collision attack, and both of them consider the original hash function's collision(not semi-free-start collision, etc). All the results are summarized in Tab. 1.

Our contributions come from three folds.

1. We introduce a new 7-round inbound differential, which extends the Sasaki and Yasuda's inbound path by 2 rounds;
2. We take advantage of the available degrees of freedom in the key to reduce the complexity of the 7-round inbound phase. This is different from the technique used by Lamberger *et al.* [17]. They also leverage the available degrees of freedom in the key. However, they use up all the degrees of freedom of the message and the key to make the inbound path hold. In our work, the freedom of the key is used in another way. The 7-round inbound phase just consumes the freedom of the message, but the complexity to compute the starting point of the outbound phase is very high. To solve the problem, we leverage the degrees of freedom of the key to significantly reduce the time complexity. This results in a 12-round *chosen-key* distinguisher of Feistel-SP block cipher.
3. Inspired by the idea that constructs collision by two blocks [26,25], we extend the *chosen-key* distinguisher to collision attacks. In our attack, the *rebound attack* is implemented in the second compression function, where the key is generated by the output of the first compression function, i.e. the chaining value. There is an interesting tradeoff between the degrees of freedom of the chaining value and the time complexity of the *rebound attack*. At last, we connect the two compression functions in the chaining value to produce a 11-round full collision.

Organization of the Paper.

Section 2 gives a brief description of Feistel-SP block cipher, some hashing modes and the *rebound attack*. Section 4 describes the new 7-round inbound differential, 12-round *chosen-key* distinguisher on Feistel-SP block ciphers and 11-round full collision attack on MMO/MP hashing mode. Then other cases of Feistel-SP block ciphers are considered in section **??**. Finally, we conclude the paper in section 7.

2 Preliminaries

In this section, the basic notations used in this paper are introduced. Then we briefly recall the properties of the Feistel block ciphers which are equipped with the SP structrues, denoted as Feistel-SP block ciphers. The hashing modes and the *rebound attack* are presented at last.

2.1 Notations

The following notations are used in this paper:

Table 1. Summary of Results for Generic Feistel-SP in Open-Key Mode

Case (N,c)†	Rounds	Time	Memory	Power	Source
(128,8)	7	–	–	known-key distinguisher	[4]
	11	2^{19}	2^{19}	known-key distinguisher	[23]
	12	2^{40}	2^{35}	chosen-key distinguisher	Section 5.1
	7	–	–	half-collision	[4]
	9	2^{27}	2^{27}	full-collision	[23]
	11	$2^{48.3}$	$2^{45.5}$	full-collision	Section 4.3
(128,4)	7	–	–	known-key distinguisher	[4]
	11	2^{12}	2^{12}	known-key distinguisher	[24]
	12	2^{36}	$2^{38.9}$	chosen-key distinguisher	Section 5.1
	7	–	–	half-collision	[4]
	9	2^{24}	2^{24}	full-collision	[23]
	11	$2^{45.5}$	$2^{45.4}$	full-collision	Section 5.1
(64,8)	7	–	–	known-key distinguisher	[4]
	9	2^{19}	2^{19}	known-key distinguisher	[24]
	7	–	–	half-collision	[4]
	7	2^{24}	2^{24}	full-collision	[23]
(64,4)	7	–	–	known-key distinguisher	[4]
	11	2^{11}	2^{11}	known-key distinguisher	[24]
	12	2^{20}	$2^{20.8}$	chosen-key distinguisher	Section 5.2
	7	–	–	half-collision	[4]
	9	2^{16}	2^{16}	full-collision	[23]
	11	$2^{24.4}$	$2^{24.4}$	full-collision	Section 5.2

†: (N,c) denote different cases of Feistel-SP block cipher, described in section 2.

N	The block length of the Feistel-SP cipher (in bits)
n	The size of the input and output of the round function, so that $n = N/2$
c	The size of an S-box in bits
r	The number of S-box sequences, so that $r = n/c$ in the Feistel-SP cipher.
X_r	the state after the key addition layer of the rth round
Y_r	the state after the substitution transformation layer of the rth round
Z_r	the state after the diffusion layer of the rth round
k_r	the subkey used in the rth round
$X[i]$	the ith byte of a bit string X, where the left most byte is $X[1]$
ΔX	the difference of X and X'
$\oplus$, $\wedge$, $\vee$	bitwise exclusive OR (XOR), AND, OR
$\bigcup$	the union of sets
$\lvert A \rvert$	the size of the set A
$\mathbf{0}$	A state where all bytes are non-active
$\mathbf{1}$	A state where only one byte of the predetermined jth position is active
$P(\mathbf{1})$	The output state of the permutation layer, when the input state is $\mathbf{1}$
$\mathbf{F}$	A state where all bytes are active

2.2 Feistel-SP block ciphers

Isobe and Shibutani [14] classify the Feistel block ciphers into three types, called *Feistel*-1/2/3. *Feistel-3* is also called Feistel-SP block cipher [24,23], which usually adopts 128-bit or 64-bit blocks and use 8-bit or 4-bit S-boxes. As introduced in [24,23], Feistel-SP block ciphers analyzed in this paper are classified as cases (N,c)=(128,8),(128,4),(64,8) and (64,4).

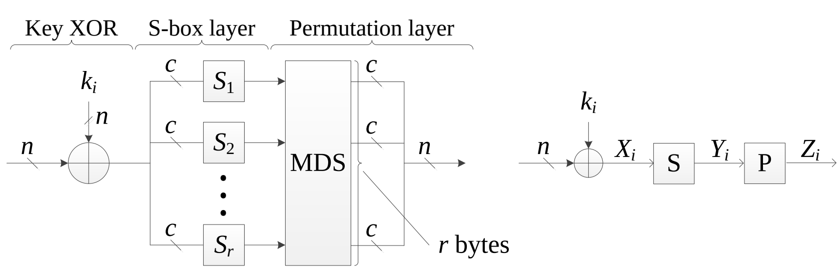

Fig. 1. Left: Detailed description of the SP round funtion **Right:**Simplified one

SP Round Functions.

As depicted in Fig.1, the SP-round function is composed of the following three elements.

– **Key XOR:** The input of the round function is XORed with a round key k_i.
– **S-box layer(S):** Each of the c-bit is substituted by S-box. All the r S-boxes $S_1, S_2, ..., S_r$ are designed to be resistant to differential and linear cryptanalyses, like the ones used in AES [6]. Hence, if given a pair of input and output differences of one S-box, there exist paired values following the given input/output differences with a probability of approximately 2^{-1}. If exist, then the number of such paired values is approximately two.
– **Permutation layer(P):** The linear diffusion, which mixes the values by multiplying an $r \times r$ matrix P, is applied to the output of the S-box layer. We make the assumption that P is a MDS matrix[4], so that the total number of active bytes in the input and output of P is at least $r + 1$, as long as the number of active bytes is not zero.

Key Schedules Assumption.

For the four block cipher cases (N,c)=(128,8),(128,4),(64,8) and (64,4), we assume their master key sizes are equal to their state size. In this paper, our techniques are based on some relatively complex key schedules, which leads to the round keys k_5 and k_9 are statistically independent, such as the key schedules[5] of AES [6], Camellia [12], CLEFIA [13], ARIA [16], etc. For some lightweight block ciphers, such as LED-128 [11], Midori [2], where they divide the master key into k_0 and k_1, and use k_0 and k_1 in turn (some round constants will be used to avoid slide attack), our attack does not work.

2.3 Hashing Modes Using block ciphers

A hash function is expected to accept almost arbitrary long message inputs. The popular Merkle-Damgård [7,19] domain extension helps us iteratively applying the compression function. Let $f(h_i, m_i)$ denote such a compression function accepting as input a message block m_i and a chaining input h_i, where h_0 is a predefined intial value. First, the input message m is padded to be a multiple of the message block length and separated into $m_0||m_1|| \cdots ||m_{L-1}$. Then, all the message blocks are iteratively processed by $h_i = f(h_{i-1}, m_{i-1})$ for $i = 1, 2, \cdots, L$. Finally, h_L is output as a hash value of m.

In paper [21], Preneel *et al.* considered a series of compression functions built from a block cipher and proved that 12 modes are secure. Matyas-Meyer-Oseas(MMO) and Miyaguchi-Preneel(MP) modes, which provide efficient ways

[4] The matching part $\mathbf{1} \rightarrow P(\mathbf{1}) \rightarrow S \leftarrow P^{-1}(\mathbf{1}) \leftarrow \mathbf{1}$ used in the inbound phase of section 4.2 requires that the active bytes positions in $P(\mathbf{1})$ and $P^{-1}(\mathbf{1})$ are the same, it is not always true if P is not a MDS matrix.

[5] It is because in section 4.2, we need the XOR of some bytes between the subkeys k_5 and k_9 should equal some special values which could be achieved by the assumed key schedules

to construct a compression function from a block cipher, are among the 12 secure schemes. Given a block cipher E and a key K, we denote its encryption algorithm as E_K. The MMO compression function computes h_i by

$$h_i = f(h_{i-1}, (m_{i-1})) = E_{h_{i-1}}(m_{i-1}) \oplus m_{i-1}, \tag{1}$$

where m_{i-1} is a message block and h_{i-1} is the previous chaining value. While the chaining value of the Miyaguchi-Preneel mode is computed by

$$h_i = f(h_{i-1}, (m_{i-1})) = E_{h_{i-1}}(m_{i-1}) \oplus m_{i-1} \oplus h_{i-1}, \tag{2}$$

given m_{i-1} and h_{i-1}.

2.4 The Rebound Attack

The *rebound attack* is a new tool for the cryptanalysis of AES-based hash functions, which was first introduced by Mendel *et al.* in [18]. The main idea is to use the available degrees of freedom in a collision attack to efficiently fulfill the low probability parts in the middle of a truncated differential trail. The rebound attack consists of an inbound phase and a outbound phase depicted in Fig. 2, where W is an internal block cipher or permutation which is split into three subparts, then $W = W_{fw} \circ W_{in} \circ W_{bw}$.

- **Inbound phase:** The inbound phase is a meet-in-the-middle phase in W_{in}. By exploiting the degrees of freedom, the attacker can generate pairs that match the truncated differential path of W_{in} in a low time cost. The matched pairs are denoted as starting points for the outbound phase.
- **Outbound phase:** In this phase, the matched pairs of the inbound phase are computed in both forward and backward direction through W_{fw} and W_{bw} to obtain a pair that satisfy the whole differential path.

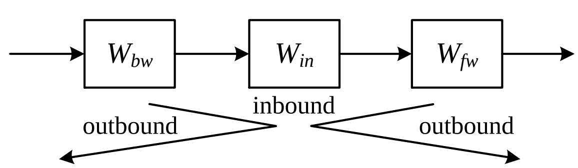

Fig. 2. The Rebound-Attack Technique

3 Sasaki and Yasuda's Work

In [24], by using rebound technique, Sasaki and Yasuda introduce 11-round *known-key* distinguisher attack on Festel-SP ciphers and 9-round collision attack on its hash mode (MMO/MP mode)[6]. All the attacks are based on a 5-round inbound differential path, as follows, depicted in Fig. 3.

$$(\mathbf{1},\mathbf{0}) \xrightarrow{4^{th}R} (\mathbf{F},\mathbf{1}) \xrightarrow{5^{th}R} (\mathbf{0},\mathbf{F}) \xrightarrow{6^{th}R} (\mathbf{F},\mathbf{0}) \xrightarrow{7^{th}R} (\mathbf{1},\mathbf{F}) \xrightarrow{8^{th}R} (\mathbf{0},\mathbf{1}).$$

Sasaki and Yasuda use the following procedures to find a starting point of the

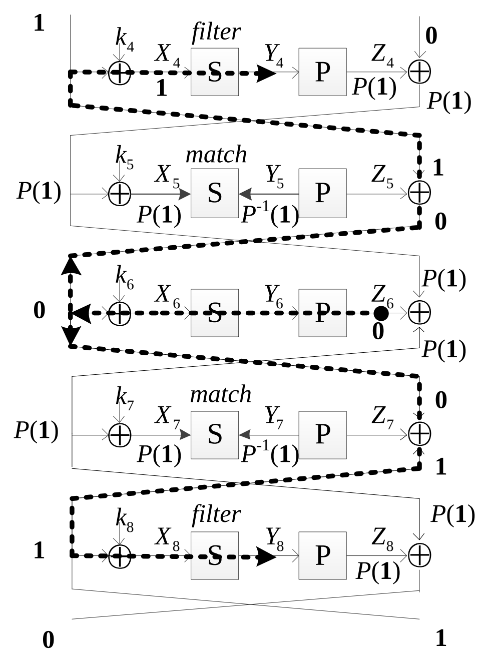

Fig. 3. The 5-round Inbound Phase Used by Sasaki and Yasuda

inbound phase:

1. Prepare DDTs for all S-boxes. Choose an active-byte position j for differential **1**

[6] For (N,c)=(64,8), it is a 9-round *known-key* distinguisher attack and 7-round collision attack

2. For all 2^c differences of ΔY_4, compute the corresponding full-byte differences after applying the (forward) permutation layer and store them in a table T. Set the difference in word ΔY_8 to be equal to ΔY_4. This guarantees that the difference in word ΔZ_6 is $\mathbf{0}$.
3. For each of the 2^c differences of ΔZ_5, compute the corresponding full-byte difference after applying the inverse permutation. For each difference stored in T, check whether we can match it with the above difference by looking up the DDTs. If a matched set of differences for ΔY_4 and ΔZ_5 is found, we can instantly obtain a matched set for ΔY_8 and ΔZ_7 by setting $\Delta Z_7 = \Delta Z_5$.
4. Now that a matched set of differences is found, we can fix word values and compute the value of word Z_6. Here the values drawn in broken lines in Fig. 3 are fixed.
   (a) Check whether or not the computed differences in ΔY_4 and ΔY_8 in step 4 and the chosen difference in $\Delta Y_4 = \Delta Y_8$ at step 2 are consistent. Namely, check the following:

$$\Delta\left[S_j\left(S_j^{-1}\left(\left(P^{-1}\left(Z_6\right)\right)[j]\right) \oplus k_6[j] \oplus Z_5[j] \oplus k_4[j]\right)\right] \stackrel{?}{=} \Delta Y_4 \qquad (3)$$

$$\Delta\left[S_j\left(S_j^{-1}\left(\left(P^{-1}\left(Z_6\right)\right)[j]\right) \oplus k_6[j] \oplus Z_7[j] \oplus k_8[j]\right)\right] \stackrel{?}{=} \Delta Y_8 \qquad (4)$$

   (b) If we find a solution for the above two equations, then it means that we have found a starting point of the inbound phase.

Then for cases (N,c)=(128,8),(128,4) and (64,4), the outbound phase for the 11-round *known-key* distinguisher attack consists of three rounds in backward direction and three rounds in the forward direction. For the 9-round collision attack, the outbound phase consists of two rounds in backward direction and two rounds in the forward direction. For case (N,c)=(64,8), the attacked rounds are reduced by 2 rounds.

4 New Attacks on Feistel-SP block ciphers: Case (N,c)=(128,8)

Sasaki and Yasuda [24] introduced a 5-round inbound differential path, based on which an 11-round *known-key* distinguisher attack on the Feistel-SP block cipher is constructed. When applying to MMO and Miyaguchi-Preneel hashing modes, they get a 9-round full-collision. In this section, we focus on the case (N,c)=(128,8), and extend the 5-round inbound differential to a 7-round one. Based it, we present a 12-round *chosen-key* distinguisher on Feistel-SP block cipher. When applying to MMO and MP hashing modes, the collision attack is extended by 2 rounds to a 11-round full-collision.

4.1 New 7-Round Inbound Differential

The differential path of the new 7-round inbound phase is

$$(\mathbf{1},\mathbf{0}) \xrightarrow{4^{th}R} (P(\mathbf{1}),\mathbf{1}) \xrightarrow{5^{th}R} (\mathbf{1},P(\mathbf{1})) \xrightarrow{6^{th}R} (\mathbf{0},\mathbf{1}) \xrightarrow{7^{th}R} (\mathbf{1},\mathbf{0}) \xrightarrow{8^{th}R} (P(\mathbf{1}),\mathbf{1}) \xrightarrow{9^{th}R} (\mathbf{1},P(\mathbf{1})) \xrightarrow{10^{th}R} (\mathbf{0},\mathbf{1}),$$

which is depicted in Fig. 4.

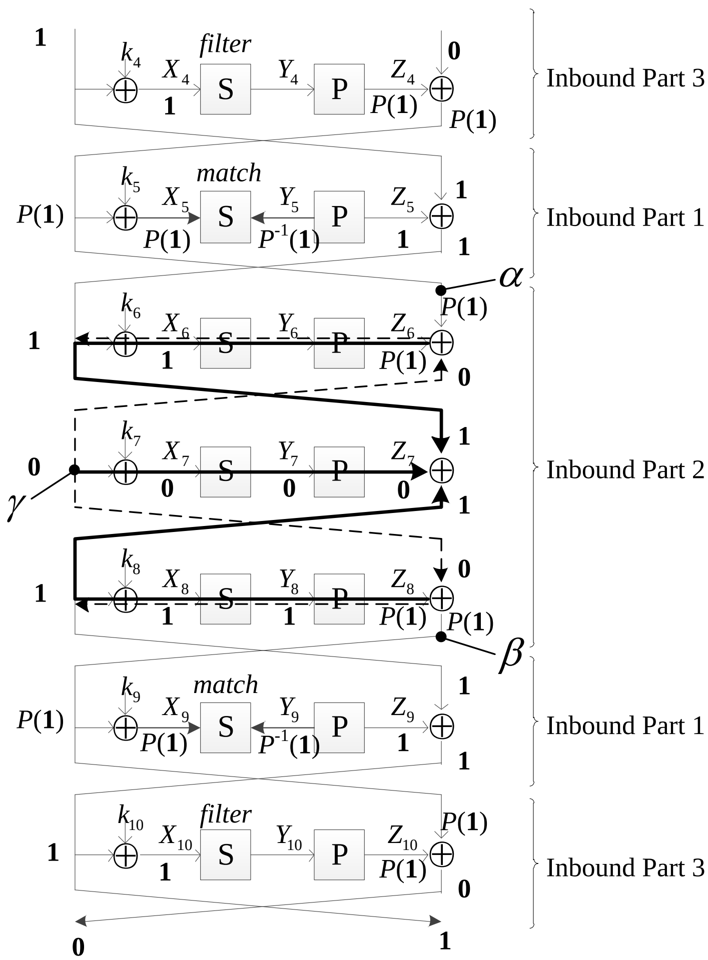

Fig. 4. 7-round inbound phase

The 7-round inbound phase is split into 3 parts, i.e. **Inbound Part 1/2/3**. In **Inbound Part 1**, the match-in-the-middle step is applied twice with active bytes $\mathbf{1} \to P(\mathbf{1}) \to S \leftarrow P^{-1}(\mathbf{1}) \leftarrow \mathbf{1}$. In **Inbound Part 2**, we follow the bold lines to formulate equation (5). The matched pairs will be connected in the middle of this part by calculating γ with equation (5). We do not need to sacrifice even a bit freedom of the key to make equation (5) hold, it has one and only one solution when given a random key. However, the nonlinear equation is hard to solve, if we calculate γ by exhaustive search, the complexity is about $2^n = 2^{64}$. Interestingly, we elaborately choose some keys to partially linearize equation (5) and make it solved much more efficiently .

$$S^{-1}(P^{-1}(\alpha \oplus \gamma)) \oplus k_6 \oplus S^{-1}(P^{-1}(\beta \oplus \gamma)) \oplus k_8 = P(S(\gamma \oplus k_7)), \qquad (5)$$

where $\alpha = X_5 \oplus k_5$, $\beta = X_9 \oplus k_9$, the subkeys k_5, k_6, k_7, k_8, k_9 are calculated by the key schedule. .

Collecting keys which help solving equation (5) easily.

For given values of X_5, X_9, $P^{-1}(\alpha \oplus \gamma) \oplus P^{-1}(\beta \oplus \gamma) = P^{-1}(X_5 \oplus k_5 \oplus \gamma) \oplus P^{-1}(X_9 \oplus k_9 \oplus \gamma) = P^{-1}(X_5 \oplus X_9) \oplus P^{-1}(k_5 \oplus k_9)$.

If we can find a master key that makes $P^{-1}(k_5 \oplus k_9)[1, 2, 3, 4, 5, 6] = P^{-1}(X_5 \oplus X_9)[1, 2, 3, 4, 5, 6]$, then $P^{-1}(\alpha \oplus \gamma)$ and $P^{-1}(\beta \oplus \gamma)$ only differ in bytes 7, 8. Hence, equation (5) is simplified as follows,

$$(0, 0, 0, 0, 0, 0, *, *) \oplus k_6 \oplus k_8 = P \circ S(\gamma \oplus k_7), \qquad (6)$$

which can be solved by traversing the two unknown bytes $*$, and correspondingly there are 2^{16} possible values of γ. Then we use equation (5) to uniquely determine the right connection value γ. The time complexity is 2^{16}. Then the following observation is easy to achieve.

Observation 1 *For a given 64-bit value $P^{-1}(X_5 \oplus X_9)$ and a 6-element-array $1 \le i_1 \le i_2 \le i_3 \le i_4 \le i_5 \le i_6 \le 8$, there are about $2^{128-48} = 2^{80}$ master keys that form a key set to ensure equation (7). There are $C_8^6 = 28$ key sets corresponding to different 6-element-arrays under a given $P^{-1}(X_5 \oplus X_9)$. We denote the union of the 28 key sets as Ukey set indexed by $P^{-1}(X_5 \oplus X_9)$, whose size is about $2^{84.8}$.*

$$P^{-1}(k_5 \oplus k_9)[i_1, i_2, i_3, i_4, i_5, i_6] = P^{-1}(X_5 \oplus X_9)[i_1, i_2, i_3, i_4, i_5, i_6], \qquad (7)$$

where k_5, k_9 are generated by the master key through the key schedule. For each key in the Ukey set, we can calculate γ with time complexity of 2^{16}.

In observation 1, the *Ukey* set is determined by the 64-bit value $P^{-1}(X_5 \oplus X_9)$. The values of X_5, X_9 are calculated from **Inbound Part 1**. In Algorithm 1, we first compute all the values of X_5, X_9 through all the possible matching difference pairs of $(\Delta X_5, \Delta Y_5)$ and $(\Delta X_9, \Delta Y_9)$, and store all the $P^{-1}(X_5 \oplus X_9)$ values in a table $\mathcal{T}$. Then we randomly choose a master key, and check equation (7) by every list in the table $\mathcal{T}$ to determine whether the key is in a $Ukey$ set. If the key is in one $Ukey$ set, we will efficiently calculate γ.

After that, we calculate forward and backward to **Inbound Part 3** and the pairs are filtered in this part. At last, the starting points prepared for the outbound phase are generated. The detailed attack procedures are shown in Algorithm 1.

Attack Evaluation.

In **Phase A**: it requires $r \cdot 2^{2c}$ computations and $r \cdot 2^{2c}$ memory to prepare r-many DDTs.

- In step 1, there are $r = 8$ possible positions for differential $\mathbf{1}$, $j = 1, 2, ...8$.
- In step 2, we are expected to find 2^{2c-r} matches between $\Delta X_5, \Delta Y_5$. Since 2^r solutions of (X_5, X'_5) are obtained from a match, hence we obtain 2^{2c} solutions as long as $2c \geq r$ which is true for case (N,c)=(128,8). It is similar to step 3.
- In step 4, there are about $2^{4c} \times 8 = 2^{35}$ (note there are 8 byte positions for j) values of $(P^{-1}(X_5 \oplus X_9),\ j,\ X_5, X'_5, X'_9)$ needed to be stored in a hash table $\mathcal{T}$.

In **Phase B**:

- In step 2, every item in table $\mathcal{T}$ corresponding to a different $Ukey$ set. If there exist an item in table $\mathcal{T}$ and a 6-element-array $1 \leq i_1 \leq i_2 \leq i_3 \leq i_4 \leq i_5 \leq i_6 \leq 8$, that make equation (7) hold, that means the chosen key falls into a $Ukey$ set. While there are 2^{35} items in $\mathcal{T}$ and 2^{35} $Ukey$ sets correspondingly. By observation 1, the chosen key falls into one of the 2^{35} $Ukey$ sets with probability of $2^{84.8} \times 2^{35} \times 2^{-128} = 2^{-8.2}$.
- In step 3, if we get a master key satisfy step 2, the value γ is calculated through equations (6) and (5). The time complexity is 2^{16} to traverse the two unknown bytes in equation (6).
- In step 4, all the bytes of ΔX_6 and ΔX_8 are zero except one in the same byte position j. This is because, $\Delta X_6 = S^{-1}(P^{-1}(X_5 \oplus k_5 \oplus \gamma)) \oplus S^{-1}(P^{-1}(X'_5 \oplus k_5 \oplus \gamma))$, and $P^{-1}(X_5 \oplus k_5 \oplus \gamma) \oplus P^{-1}(X'_5 \oplus k_5 \oplus \gamma) = P^{-1}(X_5 \oplus X'_5)$ where $\Delta X_5 = X_5 \oplus X'_5$ is of $P(\mathbf{1})$ differential pattern, so $P^{-1}(X_5 \oplus X'_5)$ is of $\mathbf{1}$ differential pattern. So ΔX_6 is of $\mathbf{1}$ differential pattern. So as to ΔX_8. Then ΔX_6 equals to ΔX_8 with probability of 2^{-8}. Once $\Delta X_6 = \Delta X_8$, $S^{-1}(P^{-1}(X_5 \oplus k_5 \oplus \gamma)) \oplus S^{-1}(P^{-1}(X'_5 \oplus k_5 \oplus \gamma)) = S^{-1}(P^{-1}(X_9 \oplus k_9 \oplus \gamma)) \oplus S^{-1}(P^{-1}(X'_9 \oplus k_9 \oplus \gamma))$ holds. Combining with equation (5), we get

$$S^{-1}(P^{-1}(X'_5 \oplus k_5 \oplus \gamma)) \oplus k_6 \oplus S^{-1}(P^{-1}(X'_9 \oplus k_9 \oplus \gamma)) \oplus k_8 = P(S(\gamma \oplus k_7)). \tag{10}$$

Algorithm 1 Calculate Starting Point by the 7-round Inbound Phase

Phase A: Prepare DDTs for all S-boxes.

1. Choose an active-byte position j for differential **1**.
2. **Inbound Part 1:** For 2^c differences of ΔY_4, compute the corresponding ΔX_5 after applying the (forward) permutation layer. For each of the 2^c differences of ΔZ_5, compute the corresponding full-byte difference ΔY_5 after applying the inverse permutation layer, and check whether ΔX_5 matches ΔY_5 by looking up the DDTs. If we pass the check, go to the following steps.
3. **Inbound Part 1:** For 2^c differences of ΔY_{10}, compute the corresponding ΔX_9 after applying the (forward) permutation layer. For all 2^c differences of ΔZ_9, compute the corresponding full-byte differences ΔY_9 after applying the inverse permutation layer, and check whether ΔX_9 matches ΔY_9 by looking up the DDTs. If we pass the check, go to the next step.
4. For the matched pairs $(\Delta X_5, \Delta Y_5)$ and $(\Delta X_9, \Delta Y_9)$, we get values (X_5, X_5'), (X_9, X_9') and store values $(P^{-1}(X_5 \oplus X_9),\ j,\ X_5, X_5', X_9')$ in a table $\mathcal{T}$.

Phase B:

1. Randomly choose a master key, and get all the subkeys by the key schedule.
2. Check equation (7) by the chosen key and every item in table $\mathcal{T}$. If there exist a item in table $\mathcal{T}$ and a 6-element-array $1 \leq i_1 \leq i_2 \leq i_3 \leq i_4 \leq i_5 \leq i_6 \leq 8$, that make equation (7) hold, go to the next step; else go to step 1 to choose another master key.
3. Calculate γ through equation (6) (note that the positions of the two unknown bytes may be changed corresponding to the 6-element-array determined in step 2.) and equation (5).
4. Follows the dashed lines, we calculate $\Delta X_6 = S^{-1}(P^{-1}(X_5 \oplus k_5 \oplus \gamma)) \oplus S^{-1}(P^{-1}(X_5' \oplus k_5 \oplus \gamma))$ and $\Delta X_8 = S^{-1}(P^{-1}(X_9 \oplus k_9 \oplus \gamma)) \oplus S^{-1}(P^{-1}(X_9' \oplus k_9 \oplus \gamma))$. If $\Delta X_6 = \Delta X_8$, then go to the next step; else go to step 1 to choose another master key.
5. Calculate $X_6 = S^{-1}(P^{-1}(X_5 \oplus k_5 \oplus \gamma))$ and X_6', X_8, X_8' similarly. Then calculate $X_4 = k_4 \oplus P(S(X_5)) \oplus X_6 \oplus k_6$ and X_4', X_{10}, X_{10}', similarly. Then check the following two equations. if these two hold, we get a starting point under the chosen key; else go to step 1 to choose another master key.

$$S_j(X_4[j]) \oplus S_j(X_4'[j]) \stackrel{?}{=} \Delta Y_4[j] \tag{8}$$

$$S_j(X_{10}[j]) \oplus S_j(X_{10}'[j]) \stackrel{?}{=} \Delta Y_{10}[j] \tag{9}$$

That means, the value of γ not only satisfy equation (5), but also is the solution of equation (10).

– In step 5, equation (8) and (9) are satisfied with probability of 2^{-16}.

Complexity Evaluation.

In **Phase A**, the time complexity is 2^{35} and memory complexity is 2^{35} 259-bit words to store $\mathcal{T}$.

In **Phase B**, if we choose 2^x different keys, there are $2^{x-8.2}$ left after step 2. The time complexity in step 3 is $2^{x-8.2} \times 2^{16} = 2^{x+7.8}$. The time complexity of step 4 is $2^{x-8.2}$ and there are $2^{x-8.2-8} = 2^{x-16.2}$ keys left. In step 5, time complexity is $2^{x-16.2}$ and there are $2^{x-16.2-16} = 2^{x-32.2}$ keys left. Finally, we get $2^{x-32.2}$ starting points. For $x = 32.2$, we get one starting point.

Totally, the time complexity is 2^{40} 7-round encryptions with bounded by step 3 of the **Phase B**. The memory complexity is 2^{35} 259-bit words to store $\mathcal{T}$. In addition, we need to randomly choose $2^{32.2}$ keys.

4.2 12-Round Chosen-Key Distinguisher

Outbound Phase. As shown in Fig. 5, we place the 7-round inbound phase between 4 and 10 round. The outbound phase is composed of 3-round backward direction and 2-round forward direction. In the backward, the differential is $(\mathbf{1}, \mathbf{0}) \xrightarrow{3^{th}} (\mathbf{0}, \mathbf{1}) \xrightarrow{2^{th}} (\mathbf{1}, P(\mathbf{1})) \xrightarrow{1^{th}} (P(\mathbf{1}), \mathbf{F})$. In the forward direction, the differential is $(\mathbf{0}, \mathbf{1}) \xrightarrow{11^{th}} (\mathbf{1}, \mathbf{0}) \xrightarrow{12^{th}} (\mathbf{1}, P(\mathbf{1}))$. Hence, when a starting point is given by Algorithm 1, the possible output difference of 12th round is limited to $2^{2c} = 2^{16}$ possible values.

Comparison with a Random Permutation.

Given a starting point of the 7-round inbound phase, the outbound phase produces a pair of values that has a differential form of $(P(\mathbf{1}), \mathbf{F})$ for plaintexts and ($\mathbf{1}$, P($\mathbf{1}$)) for ciphertexts with probability of 1. So the complexity to obtain such pairs is equal to finding a starting point of the 7-round inbound phases, which is 2^{40}. When compared with the generic birthday bound, namely $2^{(N-2c)/2} = 2^{56}$, our attack is faster by a factor 2^{16}.

4.3 11-round Collision Attack

Similarly to the *known-key* setting, the *chosen-key* setting also belongs to the *open-key* model in the literature. In [4] Biryukov *et al.* extended the *chosen-key* distinguisher to a related-key attack on the full AES-256 version, which makes the *chosen-key* model popular. However, these attacks have little effect on the practical use of AES. So one may wonder how practically will these *open-key* models endanger the real-world cryptography. In [24], Sasaki and Yasuda show that their *known-key* distinguishers can be converted to collision attacks on the

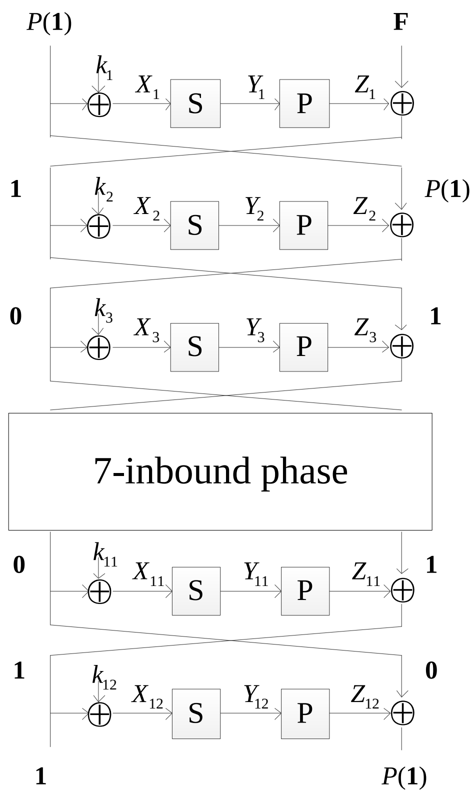

Fig. 5. 12-round chosen-key distinguisher for Feistel-SP cipher

MMO and MP hashing modes with Feistel-SP block cipher. In this section, we show that the *chosen-key* model can also be extended to collision attacks these hash functions.

Here we only discuss our attacks on the MMO mode, but all the attacks can be trivially extended to the MP mode. This is because the key addition to the hash output state used by the MP mode does not make any impact upon the output value differences.

We add two rounds on the top and two rounds at the bottom of the 7-round inbound path to construct a new 11-round rebound attack, depicted in Fig. 6. As shown in Fig. 7, we use two hashing blocks to construct a collision, where the 11-round rebound attack is used in the second compression function. The detail attack procedures are shown in Algorithm 2.

Algorithm 2 Generate 11-Round Collision for MMO Hashing Mode with Feistel-SP block cipher

---

Phase A: Carry out the **Phase A** of Algorithm 1.

Phase B:

1. As shown in Fig.8, randomly choose 2^x values M_0, compute H_1. Carry out the **Phase B** of Algorithm 1 of the 7-round inbound attack, where 2^x values of H_1 work as the chosen keys, and calculate the starting points. We get $2^{x-8.2-8-16} = 2^{x-32.2}$ starting points.
2. Then we turn to the outbound phase and calculate two rounds forward and two rounds backward. If we get a collision in H_2, the whole attack stops.

Attack Evaluation. The whole collision attack is shown in Fig. 8, we give a brief evaluation in the following.

In the **Phase B**, for step 1, we choose $x = 48.2$, and we get $2^{48.2-32.2} = 2^{16}$ starting points[7] after the **Phase B** of Algorithm 1 of the 7-round inbound attack is carried out. In section 4.2, the complexity analysis of Algorithm 1 shows that it costs 2^{40} time complexity[8] to find a starting point. So for 2^{16} starting points, the time complexity is 2^{56}. In step 2, for the outbound phase, the collision occurs when the input difference and output differences of the block cipher E_K are equal, depicted in Fig. 6. Since the input difference and output differences have the common differential pattern $(\mathbf{1}, P(\mathbf{1}))$, they are equal with probability of

[7] Note that, in the complexity evaluation of the **Phase B** of Algorithm 1, we get one starting point using $2^{32.2}$ chosen keys.

[8] Note that, this complexity evaluation from the **Phase B** of Algorithm 1 includes the time to meet the 2c-bit condition of equation (8) and (9), we do not add it here duplicately.

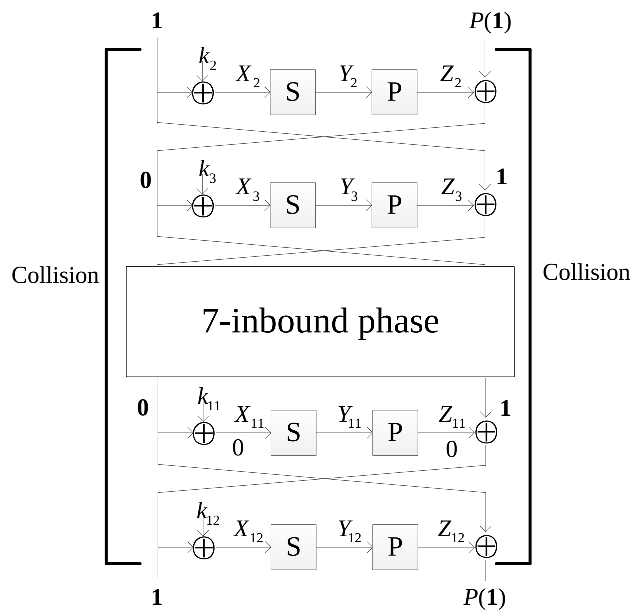

Fig. 6. 11-round collsion attack on Feistel-SP cipher

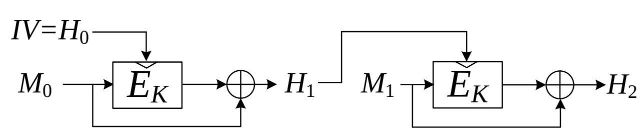

Fig. 7. MMO hashing mode with two blocks

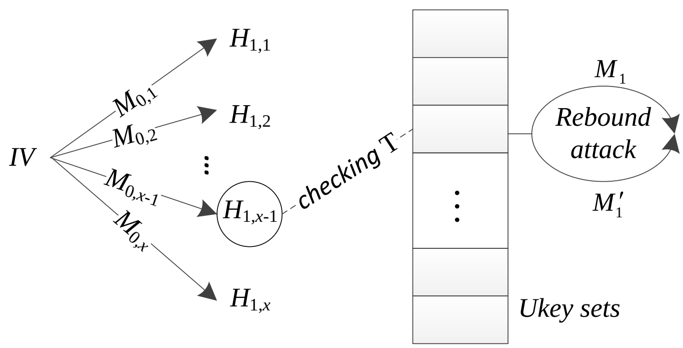

Fig. 8. The Whole Collision Attack Structure

2^{-16}. So we the 2^{16} starting points finally produce a collision in H_2. The time complexity of this step is 2^{16}.

Totally, the time complexity to find a collision in H_2 is 2^{56}. The memory complexity is dominated by the **Phase A** is 2^{35} 259-bit words, which equals to Algorithm 1. In addition, $2^{48.2}$ values of M_0 are needed.

Moreover, if we have enough parallel computing resource, we can tradeoff the memory and time complexity by using $2^{10.5}$ computing units in parallel. Then the total memory complexity is $2^{45.5}$ 259-bit words and the total time complexity is $2^{45.5+48.2} = 2^{48.3}$ hashes, while generic birthday bound is 2^{64}.

5 Attacks on Feistel-SP block ciphers: Other Cases (N,c)=(128,4),(64,8) and (64,4)

5.1 Attacks: Case (N,c)=(128,4)

For case (N,c)=(128,4), we modify the inbound phase as

$$(\mathbf{2},\mathbf{0}) \xrightarrow{4^{th}} (P(\mathbf{2}),\mathbf{2}) \xrightarrow{5^{th}} (\mathbf{2},P(\mathbf{2})) \xrightarrow{6^{th}} (\mathbf{0},\mathbf{2}) \xrightarrow{7^{th}} (\mathbf{2},\mathbf{0}) \xrightarrow{8^{th}} (P(\mathbf{2}),\mathbf{2}) \xrightarrow{9^{th}} (\mathbf{2},P(\mathbf{2})) \xrightarrow{10^{th}} (\mathbf{0},\mathbf{2}).$$

Modify the observation 1: choose an 13-element-array $1 \le i_1 \le i_2 \le i_3 \cdots \le i_{11} \le i_{12} \le i_{13} \le 16$ to construct a master key set that make equation (11) hold for a given $P^{-1}(X_5 \oplus X_9)$. There are $C_{16}^{13} = 2^{9.1}$ different key sets, whose union is also denoted as $Ukey$ set. Hence, the size of $Ukey$ is $2^{128-52+9.1} = 2^{85.1}$ for a given $P^{-1}(X_5 \oplus X_9)$.

$$P^{-1}(k_5 \oplus k_9)[i_1, i_2, \cdots, i_{12}, i_{13}] = P^{-1}(X_5 \oplus X_9)[i_1, i_2, \cdots, i_{12}, i_{13}] \quad (11)$$

Similar to Alg. 1, the match-in-the-middle step is applied twice, then we get $2^{8\times 4} \times C_{16}^2 = 2^{38.9}$ 274-bit values $(P^{-1}(X_5 \oplus X_9), i, j, X_5, X_5', X_9')$ stored in a table $\mathcal{T}'$, where i, j are active nibble positions of differential $(\mathbf{2}, \mathbf{0})$ and $1 \leq i < j \leq 16$. Then by applying **Phase B** of Alg. 1, we find a starting point by using 2^{28} random chosen keys, the time complexity is 2^{36} 7-round encryptions and the total memory cost is $2^{38.9}$ 274-bit words.

The differential of the rebound attack for 12-round *chosen-key* distinguisher is

$$(P(\mathbf{2}), \mathbf{F}) \xrightarrow{1^{th}} (\mathbf{2}, P(\mathbf{2})) \xrightarrow{2^{th}} (\mathbf{0}, \mathbf{1}) \xrightarrow{3^{th}} (\mathbf{2}, \mathbf{0}) \xrightarrow{Inbound}$$
$$(\mathbf{0}, \mathbf{2}) \xrightarrow{11^{th}} (\mathbf{2}, \mathbf{0}) \xrightarrow{12^{th}} (\mathbf{2}, P(\mathbf{2})).$$

The complexity to find a pair that matches the output differential $(\mathbf{2}, P(\mathbf{2}))$ is equal to finding a starting point of the 7-inbound phase, obviously it is faster than the generic birthday bound 2^{56}.

The collision attack is similar to Alg. 2, the differential of the rebound attack start from the second round of the 12-round differential. After tradeoff of the memory and time, the time complexity is $2^{45.5}$ hashes, the memory cost is $2^{45.4}$ 274-bit words, and 2^{44} messages are needed. While, the generic birthday bound is 2^{64}.

5.2 Attacks: Case (N,c)=(64,4) and (N,c)=(64,8)

Cases (N,c)=(64,4) is similar to (128,8), the time complexity of 12-round *chosen-key* distinguisher is 2^{20}, the memory cost is $2^{20.8}$ 131-bit words, $2^{12.2}$ random chosen keys are needed. While the generic birthday bound is $2^{(64-8)/2} = 2^{28}$. For 11-round collision attack, the time complexity is $2^{24.4}$, the memory cost is $2^{24.4}$ 131-bit words, $2^{20.2}$ messages are needed. While the generic birthday bound is $2^{64/2} = 2^{32}$.

In the case (N,c)=(64,8), the time to find a starting point of the 7-round inbound phase is about 2^{32}. So the 7-round inbound phase can not be used in this case.

6 Experiment For Case (N,c)=(128,8)

Due to the lack of Feistel-SP block cipher for case (N,c)=(128,8), we modify Camellia [12] block cipher for experiment. As we know, the linear permutation of Camellia is not a MDS matrix, so we replace it by the MDS matrix P showed in equation (12) borrowed from block cipher Khazad [3], then we remove the FL/FL^{-1} layer and make the other modules of Camellia unchanged. Note that the the linear permutation in the key schedule is also replaced by the MDS matrix. We call the new block cipher as Camellia-MDS, whose C++ code is listed in Appendix A[9]. Our experiment works on 12-round reduced Camellia-MDS with 128-bit key.

$$P = \begin{pmatrix} 0x01 & 0x03 & 0x04 & 0x05 & 0x06 & 0x08 & 0x0B & 0x07 \\ 0x03 & 0x01 & 0x05 & 0x04 & 0x08 & 0x06 & 0x07 & 0x0B \\ 0x04 & 0x05 & 0x01 & 0x03 & 0x0B & 0x07 & 0x06 & 0x08 \\ 0x05 & 0x04 & 0x03 & 0x01 & 0x07 & 0x0B & 0x08 & 0x06 \\ 0x06 & 0x08 & 0x0B & 0x07 & 0x01 & 0x03 & 0x04 & 0x05 \\ 0x08 & 0x06 & 0x07 & 0x0B & 0x03 & 0x01 & 0x05 & 0x04 \\ 0x0B & 0x07 & 0x06 & 0x08 & 0x04 & 0x05 & 0x01 & 0x03 \\ 0x07 & 0x0B & 0x08 & 0x06 & 0x05 & 0x04 & 0x03 & 0x01 \end{pmatrix} \qquad (12)$$

We give a experiment for 12-round *chosen-key* distinguisher of section 4.2 to find a pair that matches the 12-round differential pattern of Fig. 5 by using our Alg 1. We get a pair of plaintexts:

$$P_1 = (1f\ 17\ 7f\ 72\ 7a\ f5\ 37\ 53,\ 5f\ f4\ d9\ 23\ 59\ e0\ e6\ 75),$$

$$P_2 = (8a\ b5\ 11\ 89\ 23\ 29\ 49\ 9f,\ a1\ 9e\ 90\ 58\ 02\ e8\ fa\ 25),$$

under $key = (69\ e4\ 4a\ 60\ 1e\ ea\ 50\ 20,\ 0a\ 3b\ 81\ ae\ ad\ 3a\ 79\ bc)$ (all the numbers are in hexadecimal). The corresponding differential of the 12-round reduced Camellia-MDS is listed in Tab. 2, which follows the differential pattern of Fig. 5. The birthday complexity to find this kind of plaintext pair is 2^{56} 12-round encryptions. We do not give a collision attack experiment because the complexity is infeasible under our computation resource.

7 Conclusion

In this paper, we give an answer to the open problem proposed by Sasaki in [22] and prove *chosen-key* scenario works well in the study of Feistel schemes. By leveraging the *rebound-attack* technique and the available degrees of freedom in the key, we introduce 11-round collision attacks on two-block MMO/MP hash functions with Feistel-SP block ciphers. These improve previous best works by two rounds. Besides, 12-round *chosen-key* distinguishers are also presented.

Due to the development of industry, the lightweight cryptography applied to resource-restricted environment becomes more and more popular. If one needs

[9] The code is modified from Camellia source code [1], we remove the FL/FL^{-1} function and replace Camellia linear permutation by MDS matrix.

Table 2. Differential of the Experiment Pair for 12-round *chosen-key* distinguisher

95 a2 6e fb 59 dc 7e cc	fe 6a 49 7b 5b 08 1c 50
32 00 00 00 00 00 00 00	95 a2 6e fb 59 dc 7e cc
00 00 00 00 00 00 00 00	32 00 00 00 00 00 00 00
32 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
02 06 08 0a 0c 10 16 0e	32 00 00 00 00 00 00 00
a9 00 00 00 00 00 00 00	02 06 08 0a 0c 10 16 0e
00 00 00 00 00 00 00 00	a9 00 00 00 00 00 00 00
a9 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
02 06 08 0a 0c 10 16 0e	a9 00 00 00 00 00 00 00
51 00 00 00 00 00 00 00	02 06 08 0a 0c 10 16 0e
00 00 00 00 00 00 00 00	51 00 00 00 00 00 00 00
51 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
a2 fb b2 10 eb 79 82 49	51 00 00 00 00 00 00 00

†: all the numbers are in hexadecimal.

both block cipher and hash function, then using a block cipher to construct a hash function can minimize the design and implementation cost. So the security analysis of these applications is immediately needed. This paper presents some results on the generic Feistel-SP block cipher used in hashing mode. However, it is far from enough. There are many works needed to be done, such as analysis on hash function with SPN block cipher, or many standardized primitives.

8 Acknowledgments

We would like to thank anonymous reviewers of FSE 2016 for their very helpful comments on this paper. We would like to thank Jian Guo and Yanzhao Shen for their fruitful discussion on this paper.

References

1. Camellia source code. Https://tls.mbed.org/camellia-source-code
2. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: A block cipher for low energy. In: Advances in Cryptology–ASIACRYPT 2015, pp. 411–436. Springer (2015)
3. Barreto, P., Rijmen, V.: The khazad legacy-level block cipher. Primitive submitted to NESSIE 97 (2000)
4. Biryukov, A., Khovratovich, D., Nikolić, I.: Distinguisher and related-key attack on the full aes-256. In: Advances in Cryptology-CRYPTO 2009, pp. 231–249. Springer (2009)
5. Coppersmith, D.: The data encryption standard (des) and its strength against attacks. IBM journal of research and development 38(3), 243–250 (1994)

6. Daemen, J., Rijmen, V.: Aes proposal: Rijndael (1998)
7. Damgård, I.B.: A design principle for hash functions. In: Advances in Cryptology-CRYPTO89 Proceedings. pp. 416–427. Springer (1990)
8. Derbez, P., Fouque, P.A., Jean, J.: Faster chosen-key distinguishers on reduced-round aes. In: Progress in Cryptology-INDOCRYPT 2012, pp. 225–243. Springer (2012)
9. Feistel, H., Notz, W., Smith, J.L., et al.: Some cryptographic techniques for machine-to-machine data communications. Proceedings of the IEEE 63(11), 1545–1554 (1975)
10. Guo, J., Jean, J., Nikolić, I., Sasaki, Y.: Meet-in-the-middle attacks on generic feistel constructions. In: Advances in Cryptology-ASIACRYPT 2014, pp. 458–477. Springer (2014)
11. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The led block cipher. In: Cryptographic Hardware and Embedded Systems–CHES 2011, pp. 326–341. Springer (2011)
12. International Organization for Standardization(ISO): International Standard-ISO/IEC 18033-3, Information technology-Security techniques-Encryption algorithms -Part 3: Block ciphers (2010)
13. International Standardization of Organization (ISO): International Standard-ISO/IEC 29192-2, Information technology-Security techniques-Lightweight cryptography -Part 2: Block ciphers (2011)
14. Isobe, T., Shibutani, K.: Generic key recovery attack on feistel scheme. In: Advances in Cryptology-ASIACRYPT 2013, pp. 464–485. Springer (2013)
15. Knudsen, L.R., Rijmen, V.: Known-key distinguishers for some block ciphers. In: Advances in Cryptology-ASIACRYPT 2007. pp. 315–324. Springer (2007)
16. Kwon, D., Kim, J., Park, S., Sung, S.H., Sohn, Y., Song, J.H., Yeom, Y., Yoon, E.J., Lee, S., Lee, J., et al.: New block cipher: Aria. In: Information Security and Cryptology-ICISC 2003, pp. 432–445. Springer (2004)
17. Lamberger, M., Mendel, F., Rechberger, C., Rijmen, V., Schläffer, M.: Rebound distinguishers: Results on the full whirlpool compression function. In: Advances in Cryptology-ASIACRYPT 2009, pp. 126–143. Springer (2009)
18. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The rebound attack: Cryptanalysis of reduced whirlpool and grøstl. In: Fast Software Encryption. pp. 260–276. Springer (2009)
19. Merkle, R.C.: One way hash functions and des. In: Advances in Cryptology-CRYPTO89 Proceedings. pp. 428–446. Springer (1990)
20. Nikolić, I., Pieprzyk, J., Sokołowski, P., Steinfeld, R.: Known and chosen key differential distinguishers for block ciphers. In: Information Security and Cryptology-ICISC 2010, pp. 29–48. Springer (2011)
21. Preneel, B., Govaerts, R., Vandewalle, J.: Hash functions based on block ciphers: A synthetic approach. In: Advances in Cryptology-CRYPTO93. pp. 368–378. Springer (1994)
22. Sasaki, Y.: Double-sp is weaker than single-sp: rebound attacks on feistel ciphers with several rounds. In: Progress in Cryptology-INDOCRYPT 2012, pp. 265–282. Springer (2012)

23. Sasaki, Y., Emami, S., Hong, D., Kumar, A.: Improved known-key distinguishers on feistel-sp ciphers and application to camellia. In: Information Security and Privacy. pp. 87–100. Springer (2012)
24. Sasaki, Y., Yasuda, K.: Known-key distinguishers on 11-round feistel and collision attacks on its hashing modes. In: Fast Software Encryption. pp. 397–415. Springer (2011)
25. Wang, X., Yu, H.: How to break md5 and other hash functions. In: Advances in Cryptology-EUROCRYPT 2005, pp. 19–35. Springer (2005)
26. Wang, X., Yu, H., Yin, Y.L.: Efficient collision search attacks on sha-0. In: Advances in Cryptology–CRYPTO 2005. pp. 1–16. Springer (2005)

A Camellia-MDS source code

```
#include "stdlib.h"
#include "stdio.h"
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#define  CAMELLIA_ENCRYPT     1
#define  CAMELLIA_DECRYPT     0
#define  ERR_CAMELLIA_INVALID_KEY_LENGTH           -0x0024
#define  ERR_CAMELLIA_INVALID_INPUT_LENGTH         -0x0026
typedef struct
{
    int nr;                     /*!<  number of rounds  */
    uint32_t rk[68];            /*!<  CAMELLIA round keys    */
}
 camellia_context;
/* 32-bit integer manipulation macros (big endian) */
#ifndef GET_UINT32_BE
#define GET_UINT32_BE(n,b,i)                            \
{                                                       \
    (n) = ( (uint32_t) (b)[(i)    ] << 24 )             \
        | ( (uint32_t) (b)[(i) + 1] << 16 )             \
        | ( (uint32_t) (b)[(i) + 2] <<  8 )             \
        | ( (uint32_t) (b)[(i) + 3]       );            \
}
#endif
#ifndef PUT_UINT32_BE
#define PUT_UINT32_BE(n,b,i)                            \
{                                                       \
    (b)[(i)    ] = (unsigned char) ( (n) >> 24 );       \
```

```
    (b)[(i) + 1] = (unsigned char) ( (n) >> 16 );       \
    (b)[(i) + 2] = (unsigned char) ( (n) >>  8 );       \
    (b)[(i) + 3] = (unsigned char) ( (n)       );       \
}
#endif
static const unsigned char SIGMA_CHARS[6][8] =
{
    { 0xa0, 0x9e, 0x66, 0x7f, 0x3b, 0xcc, 0x90, 0x8b },
    { 0xb6, 0x7a, 0xe8, 0x58, 0x4c, 0xaa, 0x73, 0xb2 },
    { 0xc6, 0xef, 0x37, 0x2f, 0xe9, 0x4f, 0x82, 0xbe },
    { 0x54, 0xff, 0x53, 0xa5, 0xf1, 0xd3, 0x6f, 0x1c },
    { 0x10, 0xe5, 0x27, 0xfa, 0xde, 0x68, 0x2d, 0x1d },
    { 0xb0, 0x56, 0x88, 0xc2, 0xb3, 0xe6, 0xc1, 0xfd }
};
static const unsigned char FSb[256] =
{
 112, 130,  44, 236, 179,  39, 192, 229, 228, 133,  87,  53, 234,  12, 174,  65,
  35, 239, 107, 147,  69,  25, 165,  33, 237,  14,  79,  78,  29, 101, 146, 189,
 134, 184, 175, 143, 124, 235,  31, 206,  62,  48, 220,  95,  94, 197,  11,  26,
 166, 225,  57, 202, 213,  71,  93,  61, 217,   1,  90, 214,  81,  86, 108,  77,
 139,  13, 154, 102, 251, 204, 176,  45, 116,  18,  43,  32, 240, 177, 132, 153,
 223,  76, 203, 194,  52, 126, 118,   5, 109, 183, 169,  49, 209,  23,   4, 215,
  20,  88,  58,  97, 222,  27,  17,  28,  50,  15, 156,  22,  83,  24, 242,  34,
 254,  68, 207, 178, 195, 181, 122, 145,  36,   8, 232, 168,  96, 252, 105,  80,
 170, 208, 160, 125, 161, 137,  98, 151,  84,  91,  30, 149, 224, 255, 100, 210,
  16, 196,   0,  72, 163, 247, 117, 219, 138,   3, 230, 218,   9,  63, 221, 148,
 135,  92, 131,   2, 205,  74, 144,  51, 115, 103, 246, 243, 157, 127, 191, 226,
  82, 155, 216,  38, 200,  55, 198,  59, 129, 150, 111,  75,  19, 190,  99,  46,
 233, 121, 167, 140, 159, 110, 188, 142,  41, 245, 249, 182,  47, 253, 180,  89,
 120, 152,   6, 106, 231,  70, 113, 186, 212,  37, 171,  66, 136, 162, 141, 250,
 114,   7, 185,  85, 248, 238, 172,  10,  54,  73,  42, 104,  60,  56, 241, 164,
  64,  40, 211, 123, 187, 201,  67, 193,  21, 227, 173, 244, 119, 199, 128, 158
};
static const unsigned char FSb2[256] =
{
 224,   5,  88, 217, 103,  78, 129, 203, 201,  11, 174, 106, 213,  24,  93, 130,
  70, 223, 214,  39, 138,  50,  75,  66, 219,  28, 158, 156,  58, 202,  37, 123,
  13, 113,  95,  31, 248, 215,  62, 157, 124,  96, 185, 190, 188, 139,  22,  52,
  77, 195, 114, 149, 171, 142, 186, 122, 179,   2, 180, 173, 162, 172, 216, 154,
  23,  26,  53, 204, 247, 153,  97,  90, 232,  36,  86,  64, 225,  99,   9,  51,
 191, 152, 151, 133, 104, 252, 236,  10, 218, 111,  83,  98, 163,  46,   8, 175,
```

```
  40, 176, 116, 194, 189,  54,  34,  56, 100,  30,  57,  44, 166,  48, 229,  68,
 253, 136, 159, 101, 135, 107, 244,  35,  72,  16, 209,  81, 192, 249, 210, 160,
  85, 161,  65, 250,  67,  19, 196,  47, 168, 182,  60,  43, 193, 255, 200, 165,
  32, 137,   0, 144,  71, 239, 234, 183,  21,   6, 205, 181,  18, 126, 187,  41,
  15, 184,   7,   4, 155, 148,  33, 102, 230, 206, 237, 231,  59, 254, 127, 197,
 164,  55, 177,  76, 145, 110, 141, 118,   3,  45, 222, 150,  38, 125, 198,  92,
 211, 242,  79,  25,  63, 220, 121,  29,  82, 235, 243, 109,  94, 251, 105, 178,
 240,  49,  12, 212, 207, 140, 226, 117, 169,  74,  87, 132,  17,  69,  27, 245,
 228,  14, 115, 170, 241, 221,  89,  20, 108, 146,  84, 208, 120, 112, 227,  73,
 128,  80, 167, 246, 119, 147, 134, 131,  42, 199,  91, 233, 238, 143,   1,  61
};
static const unsigned char FSb3[256] =
{
  56,  65,  22, 118, 217, 147,  96, 242, 114, 194, 171, 154, 117,   6,  87, 160,
 145, 247, 181, 201, 162, 140, 210, 144, 246,   7, 167,  39, 142, 178,  73, 222,
  67,  92, 215, 199,  62, 245, 143, 103,  31,  24, 110, 175,  47, 226, 133,  13,
  83, 240, 156, 101, 234, 163, 174, 158, 236, 128,  45, 107, 168,  43,  54, 166,
 197, 134,  77,  51, 253, 102,  88, 150,  58,   9, 149,  16, 120, 216,  66, 204,
 239,  38, 229,  97,  26,  63,  59, 130, 182, 219, 212, 152, 232, 139,   2, 235,
  10,  44,  29, 176, 111, 141, 136,  14,  25, 135,  78,  11, 169,  12, 121,  17,
 127,  34, 231,  89, 225, 218,  61, 200,  18,   4, 116,  84,  48, 126, 180,  40,
  85, 104,  80, 190, 208, 196,  49, 203,  42, 173,  15, 202, 112, 255,  50, 105,
   8,  98,   0,  36, 209, 251, 186, 237,  69, 129, 115, 109, 132, 159, 238,  74,
 195,  46, 193,   1, 230,  37,  72, 153, 185, 179, 123, 249, 206, 191, 223, 113,
  41, 205, 108,  19, 100, 155,  99, 157, 192,  75, 183, 165, 137,  95, 177,  23,
 244, 188, 211,  70, 207,  55,  94,  71, 148, 250, 252,  91, 151, 254,  90, 172,
  60,  76,   3,  53, 243,  35, 184,  93, 106, 146, 213,  33,  68,  81, 198, 125,
  57, 131, 220, 170, 124, 119,  86,   5,  27, 164,  21,  52,  30,  28, 248,  82,
  32,  20, 233, 189, 221, 228, 161, 224, 138, 241, 214, 122, 187, 227,  64,  79
};
static const unsigned char FSb4[256] =
{
 112,  44, 179, 192, 228,  87, 234, 174,  35, 107,  69, 165, 237,  79,  29, 146,
 134, 175, 124,  31,  62, 220,  94,  11, 166,  57, 213,  93, 217,  90,  81, 108,
 139, 154, 251, 176, 116,  43, 240, 132, 223, 203,  52, 118, 109, 169, 209,   4,
  20,  58, 222,  17,  50, 156,  83, 242, 254, 207, 195, 122,  36, 232,  96, 105,
 170, 160, 161,  98,  84,  30, 224, 100,  16,   0, 163, 117, 138, 230,   9, 221,
 135, 131, 205, 144, 115, 246, 157, 191,  82, 216, 200, 198, 129, 111,  19,  99,
 233, 167, 159, 188,  41, 249,  47, 180, 120,   6, 231, 113, 212, 171, 136, 141,
 114, 185, 248, 172,  54,  42,  60, 241,  64, 211, 187,  67,  21, 173, 119, 128,
 130, 236,  39, 229, 133,  53,  12,  65, 239, 147,  25,  33,  14,  78, 101, 189,
```

```
 184, 143, 235, 206,  48,  95, 197,  26, 225, 202,  71,  61,   1, 214,  86,  77,
  13, 102, 204,  45,  18,  32, 177, 153,  76, 194, 126,   5, 183,  49,  23, 215,
  88,  97,  27,  28,  15,  22,  24,  34,  68, 178, 181, 145,   8, 168, 252,  80,
 208, 125, 137, 151,  91, 149, 255, 210, 196,  72, 247, 219,   3, 218,  63, 148,
  92,   2,  74,  51, 103, 243, 127, 226, 155,  38,  55,  59, 150,  75, 190,  46,
 121, 140, 110, 142, 245, 182, 253,  89, 152, 106,  70, 186,  37,  66, 162, 250,
   7,  85, 238,  10,  73, 104,  56, 164,  40, 123, 201, 193, 227, 244, 199, 158
};
#define SBOX1(n) FSb[(n)]
#define SBOX2(n) FSb2[(n)]
#define SBOX3(n) FSb3[(n)]
#define SBOX4(n) FSb4[(n)]
static const unsigned char shifts[2][4][4] =
{
    {
        { 1, 1, 1, 1 }, /* KL */
        { 0, 0, 0, 0 }, /* KR */
        { 1, 1, 1, 1 }, /* KA */
        { 0, 0, 0, 0 }  /* KB */
    },
    {
        { 1, 0, 1, 1 }, /* KL */
        { 1, 1, 0, 1 }, /* KR */
        { 1, 1, 1, 0 }, /* KA */
        { 1, 1, 0, 1 }  /* KB */
    }
};
static const signed char indexes[2][4][20] =
{
    {
        {  0,  1,  2,  3,  8,  9, 10, 11, 38, 39,
          36, 37, 23, 20, 21, 22, 27, -1, -1, 26 }, /* KL -> RK */
        { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
          -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 }, /* KR -> RK */
        {  4,  5,  6,  7, 12, 13, 14, 15, 16, 17,
          18, 19, -1, 24, 25, -1, 31, 28, 29, 30 }, /* KA -> RK */
        { -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
          -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 }  /* KB -> RK */
    },
    {
        {  0,  1,  2,  3, 61, 62, 63, 60, -1, -1,
```

```
          -1, -1, 27, 24, 25, 26, 35, 32, 33, 34 }, /* KL -> RK */
        { -1, -1, -1, -1,  8,  9, 10, 11, 16, 17,
          18, 19, -1, -1, -1, -1, 39, 36, 37, 38 }, /* KR -> RK */
        { -1, -1, -1, -1, 12, 13, 14, 15, 58, 59,
          56, 57, 31, 28, 29, 30, -1, -1, -1, -1 }, /* KA -> RK */
        {  4,  5,  6,  7, 65, 66, 67, 64, 20, 21,
          22, 23, -1, -1, -1, -1, 43, 40, 41, 42 }  /* KB -> RK */
    }
};
static const signed char transposes[2][20] =
{
    {
        21, 22, 23, 20,
        -1, -1, -1, -1,
        18, 19, 16, 17,
        11,  8,  9, 10,
        15, 12, 13, 14
    },
    {
        25, 26, 27, 24,
        29, 30, 31, 28,
        18, 19, 16, 17,
        -1, -1, -1, -1,
        -1, -1, -1, -1
    }
};
/*Shift macro for 128 bit strings with rotation smaller than 32 bits*/
#define ROTL(DEST, SRC, SHIFT)                                          \
{                                                                       \
    (DEST)[0] = (SRC)[0] << (SHIFT) ^ (SRC)[1] >> (32 - (SHIFT));   \
    (DEST)[1] = (SRC)[1] << (SHIFT) ^ (SRC)[2] >> (32 - (SHIFT));   \
    (DEST)[2] = (SRC)[2] << (SHIFT) ^ (SRC)[3] >> (32 - (SHIFT));   \
    (DEST)[3] = (SRC)[3] << (SHIFT) ^ (SRC)[0] >> (32 - (SHIFT));   \
}
//remove the FL/FL^-1 layer
/*
#define FL(XL, XR, KL, KR)                                              \
{                                                                       \
    (XR) = ((((XL) & (KL)) << 1) | (((XL) & (KL)) >> 31)) ^ (XR);   \
    (XL) = ((XR) | (KR)) ^ (XL);                                    \
}
```

```
#define FLInv(YL, YR, KL, KR)                                         \
{                                                                     \
    (YL) = ((YR) | (KR)) ^ (YL);                                      \
    (YR) = ((((YL) & (KL)) << 1) | (((YL) & (KL)) >> 31)) ^ (YR);    \
}
*/
#define SHIFT_AND_PLACE(INDEX, OFFSET)                        \
{                                                             \
    TK[0] = KC[(OFFSET) * 4 + 0];                             \
    TK[1] = KC[(OFFSET) * 4 + 1];                             \
    TK[2] = KC[(OFFSET) * 4 + 2];                             \
    TK[3] = KC[(OFFSET) * 4 + 3];                             \
                                                              \
    for( i = 1; i <= 4; i++ )                                 \
        if( shifts[(INDEX)][(OFFSET)][i -1] )                 \
            ROTL(TK + i * 4, TK, ( 15 * i ) % 32);            \
                                                              \
    for( i = 0; i < 20; i++ )                                 \
        if( indexes[(INDEX)][(OFFSET)][i] != -1 ) {           \
            RK[indexes[(INDEX)][(OFFSET)][i]] = TK[ i ];      \
        }                                                     \
}
// p[][] is a MDS matrix borrowed from block cipher Khazad.
int p[8][8]={
{0x01,0x03,0x04,0x05,0x06,0x08,0x0B,0x07},
{0x03,0x01,0x05,0x04,0x08,0x06,0x07,0x0B},
{0x04,0x05,0x01,0x03,0x0B,0x07,0x06,0x08},
{0x05,0x04,0x03,0x01,0x07,0x0B,0x08,0x06},
{0x06,0x08,0x0B,0x07,0x01,0x03,0x04,0x05},
{0x08,0x06,0x07,0x0B,0x03,0x01,0x05,0x04},
{0x0B,0x07,0x06,0x08,0x04,0x05,0x01,0x03},
{0x07,0x0B,0x08,0x06,0x05,0x04,0x03,0x01}
};
#define xtime(a) ((a&0x80)?(((a<<1)^0x1d)&0xff):((a<<1)&0xff))
#define xtime1(a) (a)
#define xtime3(a) (xtime(a)^a)
#define xtime4(a) (xtime(xtime(a)))
#define xtime5(a) (xtime(xtime(a))^a)
#define xtime6(a) (xtime(xtime(a))^xtime(a))
#define xtime7(a) (xtime6(a)^a)
#define xtime8(a) (xtime(xtime(xtime(a))))
```

```
#define xtimeb(a) (xtime8(a)^xtime3(a))
void permut(uint32_t x0, uint32_t x1, uint32_t z[2])
{
unsigned char  a[8];
PUT_UINT32_BE(x0,a,0);
PUT_UINT32_BE(x1,a,4);
unsigned char in[8];
for(int i=0;i<8;i++)in[i]=a[i];
 a[0]=xtime1(in[0])^xtime3(in[1])^xtime4(in[2])^xtime5(in[3])
         ^xtime6(in[4])^xtime8(in[5])^xtimeb(in[6])^xtime7(in[7]);
 a[1]=xtime3(in[0])^xtime1(in[1])^xtime5(in[2])^xtime4(in[3])
         ^xtime8(in[4])^xtime6(in[5])^xtime7(in[6])^xtimeb(in[7]);
 a[2]=xtime4(in[0])^xtime5(in[1])^xtime1(in[2])^xtime3(in[3])
        ^xtimeb(in[4])^xtime7(in[5])^xtime6(in[6])^xtime8(in[7]);
 a[3]=xtime5(in[0])^xtime4(in[1])^xtime3(in[2])^xtime1(in[3])
        ^xtime7(in[4])^xtimeb(in[5])^xtime8(in[6])^xtime6(in[7]);

 a[4]=xtime6(in[0])^xtime8(in[1])^xtimeb(in[2])^xtime7(in[3])
        ^xtime1(in[4])^xtime3(in[5])^xtime4(in[6])^xtime5(in[7]);
 a[5]=xtime8(in[0])^xtime6(in[1])^xtime7(in[2])^xtimeb(in[3])
        ^xtime3(in[4])^xtime1(in[5])^xtime5(in[6])^xtime4(in[7]);
 a[6]=xtimeb(in[0])^xtime7(in[1])^xtime6(in[2])^xtime8(in[3])
        ^xtime4(in[4])^xtime5(in[5])^xtime1(in[6])^xtime3(in[7]);
 a[7]=xtime7(in[0])^xtimeb(in[1])^xtime8(in[2])^xtime6(in[3])
        ^xtime5(in[4])^xtime4(in[5])^xtime3(in[6])^xtime1(in[7]);
uint32_t b, c;
GET_UINT32_BE(b,a,0);
GET_UINT32_BE(c,a,4);
z[0]=z[0]^b;
z[1]=z[1]^c;
}
static void camellia_feistel( const uint32_t x[2], const uint32_t k[2],
                              uint32_t z[2])
{
    uint32_t I0, I1;
    I0 = x[0] ^ k[0];
    I1 = x[1] ^ k[1];

    I0 = ((uint32_t) SBOX1((I0 >> 24) & 0xFF) << 24) |
         ((uint32_t) SBOX2((I0 >> 16) & 0xFF) << 16) |
         ((uint32_t) SBOX3((I0 >>  8) & 0xFF) <<  8) |
```

```
        ((uint32_t) SBOX4((I0      ) & 0xFF)      );
    I1 = ((uint32_t) SBOX2((I1 >> 24) & 0xFF) << 24) |
         ((uint32_t) SBOX3((I1 >> 16) & 0xFF) << 16) |
         ((uint32_t) SBOX4((I1 >>  8) & 0xFF) <<  8) |
         ((uint32_t) SBOX1((I1      ) & 0xFF)      );

permut(I0, I1, z);//replace linear permutation of Camellia by a MDS matrix
/*
    I0 ^= (I1 << 8) | (I1 >> 24);
    I1 ^= (I0 << 16) | (I0 >> 16);
    I0 ^= (I1 >> 8) | (I1 << 24);
    I1 ^= (I0 >> 8) | (I0 << 24);
    z[0] ^= I1;
    z[1] ^= I0;
*/
}

void  camellia_init(  camellia_context *ctx )
{
    memset( ctx, 0, sizeof(  camellia_context ) );
}
void  camellia_free(  camellia_context *ctx )
{
    if( ctx == NULL )
        return;
 memset( ctx, 0, sizeof(camellia_context) );

}
/* Camellia key schedule (encryption) */
int  camellia_setkey_enc(  camellia_context *ctx, const unsigned char *key,
                         unsigned int keybits )
{
    int idx;
    size_t i;
    uint32_t *RK;
    unsigned char t[64];
    uint32_t SIGMA[6][2];
    uint32_t KC[16];
    uint32_t TK[20];

    RK = ctx->rk;
```

```
memset( t, 0, 64 );
memset( RK, 0, sizeof(ctx->rk) );

switch( keybits )
{
    case 128: ctx->nr = 3; idx = 0; break;
    case 192:
    case 256: ctx->nr = 4; idx = 1; break;
    default : return(  ERR_CAMELLIA_INVALID_KEY_LENGTH );
}

for( i = 0; i < keybits / 8; ++i )
    t[i] = key[i];
/* Prepare SIGMA values */
for( i = 0; i < 6; i++ ) {
    GET_UINT32_BE( SIGMA[i][0], SIGMA_CHARS[i], 0 );
    GET_UINT32_BE( SIGMA[i][1], SIGMA_CHARS[i], 4 );
}

/*
 * Key storage in KC
 * Order: KL, KR, KA, KB
 */
memset( KC, 0, sizeof(KC) );
/* Store KL, KR */
for( i = 0; i < 8; i++ )
    GET_UINT32_BE( KC[i], t, i * 4 );
/* Generate KA */
for( i = 0; i < 4; ++i )
    KC[8 + i] = KC[i] ^ KC[4 + i];
camellia_feistel( KC + 8, SIGMA[0], KC + 10 );
camellia_feistel( KC + 10, SIGMA[1], KC + 8 );
for( i = 0; i < 4; ++i )
    KC[8 + i] ^= KC[i];
camellia_feistel( KC + 8, SIGMA[2], KC + 10 );
camellia_feistel( KC + 10, SIGMA[3], KC + 8 );
if( keybits > 128 ) {
    /* Generate KB */
    for( i = 0; i < 4; ++i )
        KC[12 + i] = KC[4 + i] ^ KC[8 + i];
```

```
        camellia_feistel( KC + 12, SIGMA[4], KC + 14 );
        camellia_feistel( KC + 14, SIGMA[5], KC + 12 );
    }
    /*Generating subkeys */
    /* Manipulating KL */
    SHIFT_AND_PLACE( idx, 0 );
    /* Manipulating KR */
    if( keybits > 128 ) {
        SHIFT_AND_PLACE( idx, 1 );
    }
    /* Manipulating KA */
    SHIFT_AND_PLACE( idx, 2 );

    /* Manipulating KB */
    if( keybits > 128 ) {
        SHIFT_AND_PLACE( idx, 3 );
    }
    /* Do transpositions */
    for( i = 0; i < 20; i++ ) {
        if( transposes[idx][i] != -1 ) {
            RK[32 + 12 * idx + i] = RK[transposes[idx][i]];
        }
    }
    return( 0 );
}
/*Camellia key schedule (decryption)*/
int  camellia_setkey_dec(  camellia_context *ctx, const unsigned char *key,
                         unsigned int keybits )
{
    int idx, ret;
    size_t i;
     camellia_context cty;
    uint32_t *RK;
    uint32_t *SK;

     camellia_init( &cty );
    /* Also checks keybits */
    if( ( ret =  camellia_setkey_enc( &cty, key, keybits ) ) != 0 )
        goto exit;
```

```
    ctx->nr = cty.nr;
    idx = ( ctx->nr == 4 );

    RK = ctx->rk;
    SK = cty.rk + 24 * 2 + 8 * idx * 2;

    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;

    for( i = 22 + 8 * idx, SK -= 6; i > 0; i--, SK -= 4 )
    {
        *RK++ = *SK++;
        *RK++ = *SK++;
    }
    SK -= 2;
    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;

exit:
    camellia_free( &cty );
    return( ret );
}
/* Camellia block encryption/decryption*/
int  camellia_crypt(  camellia_context *ctx,
                    int mode,
                    const unsigned char input[16],
                    unsigned char output[16] )
{
    int NR;
    uint32_t *RK, X[4];
    ( (void) mode );
NR = ctx->nr;
    RK = ctx->rk;
    GET_UINT32_BE( X[0], input,  0 );
    GET_UINT32_BE( X[1], input,  4 );
    GET_UINT32_BE( X[2], input,  8 );
    GET_UINT32_BE( X[3], input, 12 );
```

```
    X[0] ^= *RK++;
    X[1] ^= *RK++;
    X[2] ^= *RK++;
    X[3] ^= *RK++;
    while( NR ) {
        camellia_feistel( X, RK, X + 2 );
        RK += 2;
        camellia_feistel( X + 2, RK, X );
        RK += 2;
        camellia_feistel( X, RK, X + 2 );
        RK += 2;
camellia_feistel( X + 2, RK, X );
        RK += 2;
        camellia_feistel( X, RK, X + 2 );
        RK += 2;
        camellia_feistel( X + 2, RK, X );
        RK += 2;
NR--;
        if( NR ) {//remove the FL/FL^-1 layer
            //FL(X[0], X[1], RK[0], RK[1]);
            RK += 2;
            //FLInv(X[2], X[3], RK[0], RK[1]);
            RK += 2;
        }
    }
    X[2] ^= *RK++;
    X[3] ^= *RK++;
    X[0] ^= *RK++;
    X[1] ^= *RK++;
    PUT_UINT32_BE( X[2], output,  0 );
    PUT_UINT32_BE( X[3], output,  4 );
    PUT_UINT32_BE( X[0], output,  8 );
    PUT_UINT32_BE( X[1], output, 12 );
    return( 0 );
}
```