Finding Collisions against 4-round SHA3-384 in Practical Time

Senyang Huang[1,3], Orna Agmon Ben-Yehuda[2], Orr Dunkelman[3], and Alexander Maximov[4]

[1] Dept. of Electrical and Information Technology, Lund University, Lund, Sweden
senyang.huang@eit.lth.se**

[2] CRI, University of Haifa, Haifa, Israel
ladypine@gmail.com

[3] Dept. of Computer Science, University of Haifa, Haifa, Israel
orrd@cs.haifa.ac.il

[4] Ericsson Research, Lund, Sweden
alexander.maximov@ericsson.com

Abstract. The Keccak sponge function family, designed by Bertoni et al. in 2007, was selected by the U.S. National Institute of Standards and Technology (NIST) in 2012 as the next generation of Secure Hash Algorithm (SHA-3). Due to its theoretical and practical importance, cryptanalysis against SHA-3 has attracted an increasing attention. To the best of our knowledge, the most powerful collision attack on SHA-3 up till now is the linearisation technique proposed by Jian Guo et al. However, that technique is infeasible to work on variants with a smaller input space, such as SHA3-384.

In this work we improve previous results with three ideas which were not used in previous works against SHA-3. First, in order to reduce constraints and increase flexibility in our solutions, we use 2-block messages instead of 1-block messages. Second, we reduce the connectivity problem into a satisfiability (SAT) problem, instead of applying the linearisation technique. Finally, we consider two new non-random properties of the Keccak non-linear layer and propose an efficient deduce-and-sieve algorithm based on these properties.

The resulting collision-finding algorithm on 4-round SHA3-384 has a practical time complexity of $2^{59.64}$ (and memory complexity of $2^{45.93}$). This greatly improves the previous best known collision attack by Dinur et al., whose 2^{147} time complexity was far from being practical. Although our attack does not threaten the security margin of the SHA-3 hash function, the tools developed in this paper could be useful in future analysis of other cryptographic primitives and also in development of new and faster SAT solvers.

Keywords: SHA-3 hash function, collision attack, deduce-and-sieve algorithm, SAT solver

** This work was partially done when Senyang Huang was a post-doc researcher at the University of Haifa.

1 Introduction

Cryptographic hash functions are unkeyed primitives that accept an arbitrarily long input *message* and produce a fixed length output *hash value*, or *digest* for short. Since the time of introduction by Diffie and Hellman in their seminal paper [7] suggesting signing a hash value of a message rather than the message itself, hash functions were found to be extremely useful in variety of cryptographic protocols: in authentication (e.g., HMAC [1]), in password protection, for commitment schemes, in key exchange protocols, etc. Hence, the need for a secure and efficient hash function is great, both for real life applications and as a component of more complex constructions.

The first de-facto cryptographic hash function was MD5 [24], developed by Rivest to fix a few issues with its predecessor MD4. Later, the US National Institute of Standards in Technology (NIST) published the SHA standard [22]. After two years, SHA was updated into what was later named SHA-1 [20] to prevent some attacks that was not disclosed to the public (but was later rediscovered by Chabaud and Joux [4]). With the need for larger output sizes (as SHA-1 supports 160-bit output), NIST has decided to publish a new family of hash functions, called SHA-2, with output sizes between 224 and 512 bits [21].

In a series of papers from 2005, Wang et al. [28,29,30] succeeded in breaking a number of cryptographic functions. These fundamental works demonstrated the way to attack most of existing hash functions using several techniques and ideas: the use of modular differences (i.e., using both a XOR difference and an additive difference), the use of multi-block collisions (i.e., collisions that span over several blocks, and idea discovered independently in [2]), and introduction of the message modification technique (a method to tweak a pair of messages conforming to some differential characteristic up to a certain round, so that it satisfies the characteristic for more rounds).

These advances (together with additional results on the Merkle-Damgård hash function [6,16] which is the design all previously mentioned hash functions followed), led NIST to start a cryptographic competition for the selection of a new hash function standard. The process started in 2008, and in 2015 Keccak [11] was published as the new SHA-3 standard [19].

Keccak [11], designed by Bertoni et al., is a sponge construction. It has a 1600-bit state which is updated by XORing message blocks to the state. The number of bits that compose a message block depends on the required output size (as the capacity of the sponge function should be twice as many bits as the output size, and the remaining bits are XORed with the message block). Then, a 24-round permutation Keccak-f is applied to the state and another block is absorbed into the state, until the last block is absorbed. Finally, the internal state is updated again using Keccak-f, and some bits of the internal state are revealed as the output.

Since 2008 when Keccak was published, the analysis of both keyed mode and unkeyed mode Keccak has attracted considerable attention in academia. On the keyed mode Keccak, a cube-like attack proposed by Dinur et al. [10] and a conditional cube attack proposed by Huang et al. [13] are the most powerful tools for analysing primitives based on the Keccak sponge function.

A SAT solver has been applied on the analysis of the unkeyed mode Keccak by Morawiecki et al. to find a preimage of a hash value in the literature [17]. But the authors did not take algebraic structures of Keccak into consideration, which somehow reduces the power of the SAT solver. In our collision attack, we combine algebraic non-random characteristics with a SAT solver to make full use of its efficiency.

The purpose of a collision attack on a hash function H is to find a pair of distinct messages M and M' such that $H(M) = H(M')$. Finding a collision pair should be computationally difficult for a secure hash function. With respect to collision attacks on Keccak, Naya-Plasencia et al. reported several practical attacks in [18] on the Keccak hash function in 2011, including a collision attack on 2-round Keccak-512.[5] Dinur et al. proposed practical collision attacks on 4-round Keccak-224/256 [8] in 2012, where the authors combined a 1-round connector with a 3-round low weight trail by algebraic techniques. In 2013, the same authors constructed practical collision attacks on 3-round Keccak-384 and Keccak-512 and theoretical attacks on 4-round Keccak-384 and 5-round Keccak-256 using generalised internal differentials [9].

Currently, the most powerful tool for building a practical collision attack against the SHA-3 hash function is the linearisation technique carried out by Qiao et al. in [23], which was later improved by Song et al. in [25]. The two works were recapped into a journal version in [12] by Guo et al. In [23], Qiao et al. followed the framework proposed by Dinur et al. in [8] and extended the previous 1-round connector by one more round. In that work, the authors developed a novel algebraic technique to linearise all S-boxes in the first round. In [25], Song et al. developed a new non-full linearisation technique to save degrees of freedom in the attack. With the help of the new technique, they launched several practical collision attacks on the Keccak family, such as 5-round SHA3-224 and 5-round SHA3-256. However, this technique can not succeed to work on variants with a smaller input space, such as SHA3-384 and SHA3-512. It is because the appended conditions will consume a large amount of degrees of freedom while variants with a smaller input space can not provide sufficient degrees of freedom. Previous results of collision attacks on SHA-3 are summarised in Table 1.

Our Contributions Our work extends previous results [8] on finding collisions in SHA-3: a 3-round differential characteristic that leads to a collision is used in rounds 2–4, whereas a connecting phase is used in the first round to lead the input message pair into the input difference of the differential characteristic.

At the same time, we introduce three techniques into collision attacks on Keccak. The first, is the use of more than a single block in the colliding message pair. Namely, we noticed that many of good input differences impose conditions on the input which cannot be satisfied (as they are in the capacity part of the state). By first finding a message pair that satisfies these conditions, we levy

[5] SHA3-n differs from Keccak-n only in the padding rule.

Rounds	Target	Complexity	Reference
2	Keccak-512	Practical	[18]
4	Keccak-224	Practical	[8]
4	Keccak-256	Practical	[8]
3	Keccak-512	Practical	[9]
3	Keccak-384	Practical	[9]
4	Keccak-384	2^{147}	[9]
5	Keccak-256	2^{115}	[9]
5	SHA3-224	Practical	[23]
5	SHA3-256	Practical	[23]

Table 1. Summary of Existing Collision Attacks on SHA-3

this restriction. Not only that this step offers greater flexibility in choosing a differential characteristic, we use the first block to set some chaining value bits to values that help the connectivity step.

Our second contribution is to replace the linear connection phase that was used before in [12] with a SAT-connection phase, which is inspired by the dedicated collision attack against SHA-1 with a SAT solver aided, proposed by Stevens et al in [27]. Namely, we use SAT solvers to find message values that satisfy the required difference conditions while previous works [12,23,25] did the connection from the input difference of the characteristic to the message conditions using linearisations. Again, there are two advantages for this approach — the first, is that we gain greater flexibility in choosing the differential characteristic as now we can "connect" to a wider range of input differences. Secondly, non-linear conditions which are useful in finding collisions (i.e., fixing intermediate bits to some values) are much easier to be satisfied using this sort of tools.

The third contribution is the introduction and development of multiple detection and sieving tools for a faster and more efficient completion of internal states than applying a SAT solver directly on non-linear problems. This reduces the number of unknowns and simplifies relations, making SAT solvers more efficient by a few orders of magnitude. We introduce a *Truncated Difference Transform Table*, that for a given truncated differential transition stores possible differential transitions (i.e., if a truncated differential is followed, the table allows to efficiently find actual bit differences that were involved in the transition). We also introduce a *Fixed Value Distribution Table*, a precomputed table used to efficiently identify values that correspond to certain truncated difference transitions (just like in the original work of [3] stored in the difference distribution table also the values that correspond to the transition). The use of these two tools makes it possible to deduce information concerning pairs which we need for satisfying the differential characteristic.

We combine these ideas and produce the first practical attack that can find collisions in 4-round SHA-384. The expected running time of this attack is below 2^{60} (we remind the reader that the SHA1 collision found by [27] used about 2^{63} computation). While we implemented the attack and verified it, our best result at the moment is a 6-bit near collision, which is to date, the best known near-collision against SHA3-384.

Organisation of the paper The rest of the paper is organised as follows. In Section 2, we describe the SHA-3 hash function and properties of the Keccak round function. In Section 3, we revisit the collision attack proposed by Guo et al. The new framework of our attack is illustrated in Section 4. The methods of constructing the differential characteristic and generating the first blocks are stated in Section 5. The SAT-connection phase is described in Section 6. In Section 7, experimental results of our attack are given. Finally, we conclude the paper in Section 8.

2 Background

2.1 SHA-3 hash function

The Keccak algorithm. In this section we describe the Keccak hash function in its default version. We refer the reader to [11,19] for the complete Keccak specification.

The Keccak hash function works on a 1600-bit state A, which is simply a three-dimensional array of bits, namely $A[5][5][64]$. One-dimensional arrays $A[\ ][y][z]$, $A[x][\ ][z]$ and $A[x][y][\ ]$ are called a column, a row and a lane, respectively; a two-dimensional array $A[\ ][\ ][z]$ is called a slice, which is shown in Figure 1. The coordinates are always considered modulo 5 for x and y, and modulo 64 for z. A 1600-bit string a is converted to the state A in the following manner: the $(64(5y+x)+z)$th bit of a becomes $A[x][y][z]$.

We will also utilise a one-dimensional manner for referring to a single related bit. For example, we use $A[i]$ to represent the bit $A[\psi_0(i)][\psi_1(i)][\psi_2(i)]$, where $\psi_0(i) = \lfloor i/320 \rfloor$, $\psi_1(i) = \lfloor i/64 \rfloor \mod 5$, $\psi_2(i) = i \mod 64$, $\lfloor\ \rfloor$ is the floor function, and $0 \leq i < 1600$. We also define a function ϕ_0 such that the bit $A[i]$ is in the $\phi_0(i)$th column of the state A, where $\phi_0(i) = 64\psi_1(i) + \psi_2(i)$.

There are four different variants of the Keccak hash function, namely Keccak-224, Keccak-256, Keccak-384 and Keccak-512. For each $n \in \{224, 256, 384, 512\}$, Keccak-$n$ corresponds to the parameters r (bitrate) and $c = 2n$ (capacity), where $r + c = 1600$. The capacity c is $448, 512, 768, 1024$, respectively for Keccak-224, Keccak-256, Keccak-384 and Keccak-512. And the bitrate r is $1152, 1088, 832, 576$ with respect to the four versions.

Initially, the state is filled with zeroes and the message is split into r-bit blocks. There are two phases in the Keccak hash function. In the absorbing phase, the next r-bit message block is XORed with its first r-bit segment of the state and then the state is processed by an internal permutation that consists of 24 rounds. After all the blocks are absorbed, the squeezing phase begins. In the squeezing phase, Keccak-n iteratively returns the first r bits of the state as the output of the function with the internal permutation, until an n-bit digest is produced.

In the permutation, the round function R consists of five operations, namely, θ, ρ, π, χ and ι. The round function is defined as $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$, with the

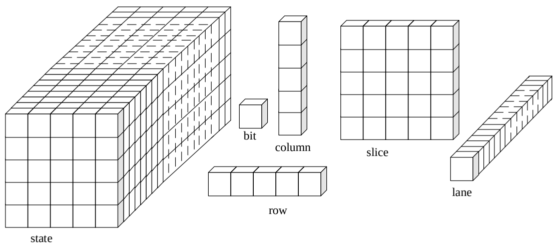

Fig. 1. Terminologies in Keccak

following sub-functions:

$$\begin{aligned}
&\theta : A[i][j][k] \leftarrow A[i][j][k] + \Sigma_{j'=0}^{4} A[i-1][j'][k] + \Sigma_{j'=0}^{4} A[i+1][j'][k-1],\\
&\rho : A[i][j] \leftarrow A[i][j] \ggg r[i,j], \quad r[i,j]\text{s are constants},\\
&\pi : A[j][2i+3j] \leftarrow A[i][j],\\
&\chi : A[i][j] \leftarrow A[i][j] + (A[i+1][j]+1)A[i+2][j][k],\\
&\iota : A[0][0] \leftarrow A[0][0] + RC_{i_r}, \quad RC_{i_r} \text{ is the } i_r\text{th round constant},
\end{aligned}$$

where $0 \le i < 5$, $0 \le j < 5$, $0 \le k < 64$ and $0 \le i_r < 24$.

The purpose of θ is to diffuse the state. In the operation θ, the bit $A[i]$ is summed up with the $\phi_1(i)$th and $\phi_2(i)$th columns of bits, where $\phi_1(i) = 64((\psi_0(i) - 1) \mod 5) + \psi_2(i)$ and $\phi_2(i) = 64((\psi_0(i) + 1) \mod 5) + ((\psi_2(i) - 1) \mod 5)$.

The operations ρ and π implement a bit-level permutation of the state. Let us denote this combined permutation by $\sigma = \pi \circ \rho$, which forms a mapping on integers $\{0, 1, \cdots, 1599\}$ such that $\sigma(i)$ is the new position of the i-th bit in the state after applying $\pi \circ \rho$. The first three linear operations θ, ρ and π will be called a *half round*, denoted as L. We rewrite the expression of L in Equation 1:

$$B[i] = A[\sigma^{-1}(i)] \oplus col[\phi_1(\sigma^{-1}(i))] \oplus col[\phi_2(\sigma^{-1}(i))]. \tag{1}$$

In Equation 1, A is the input state of L while B is the output state. $col[\phi_1(\sigma^{-1}(i))]$ and $col[\phi_2(\sigma^{-1}(i))]$ are the sums of the five bits in the $\phi_1(\sigma^{-1}(i))$th and $\phi_2(\sigma^{-1}(i))$th columns, respectively. The sum of the five bits in one column is called a *column sum*.

Padding rule. The Keccak hash function uses a multi-rate padding rule. By this rule, the original message M is appended with a single bit 1 followed by the minimum number of 0 bits and a single 1 bit such that the resulting message is of length that is a multiple of the bitrate r. Specifically, the resulting padded message is $\overline{\mathrm{M}} = M|10*1$.

In the four Keccak variants adopted by the SHA-3 standard, the message is first appended with '01', then the padding rule is applied. Namely, the resulting padded message is $\overline{\mathrm{M}} = M|0110 * 1$.

2.2 Properties of the Keccak round function

In this section we show five properties of the Keccak round function. The first property is called the *CP-kernel* equation [11]. For states in which all columns have even parity, θ is the identity: this is the *column parity kernel (CP-kernel)* property. This property has been widely used in cryptanalysis of Keccak. For example, the attacks in [13] use it to control the diffusion of cube variables.

Property 1. (**CP-kernel Equation**) For every ith and jth bits in the same column of the state A we have:

$$A[i] \oplus A[j] = B[\sigma(i)] \oplus B[\sigma(j)],$$

where A and B are the input and output states of L, respectively, and $0 \leq i, j < 1600,\ i \neq j$.

Property 1 can be easily verified through Equation 1. As the operations in the first half round are all linear, the equality also holds for differences of corresponding bits.

Before we present four differential properties of the non-linear operation χ, we first recall the definition of the *difference distribution table* (DDT) of χ. The operation χ is applied to each row of the state independently, and can be regarded as an *S-box.* In the differential cryptanalysis proposed by Biham and Shamir in [3], the DDT of an S-box counts the number of cases where the input difference of a pair is a and the output difference is b. In our case, for an input difference $a \in \mathbb{F}_2^5$ and an output difference $b \in \mathbb{F}_2^5$, the entry $\delta(a, b)$ of the DDT of the Keccak S-box S is:

$$\delta(a, b) = |\{z \in \mathbb{F}_2^5 | S(z) \oplus S(z \oplus a) = b\}|.$$

The set $\{z \in \mathbb{F}_2^5 | S(z) \oplus S(z \oplus a) = b\}$ is called a *solution set.*[6] We present an important property of the solution set as follows.

Property 2. ([5,8,11]) For every $a, b \in \mathbb{F}_2^5$, the solution set of the Keccak S-box, $\{z \in \mathbb{F}_2^5 | S(z) \oplus S(z \oplus a) = b\}$ forms an affine subspace of $\mathbb{F}_2^5$.

In Property 3 and Property 4, we will show that for some special output differences of the Keccak S-box, the input difference should follow certain conditions. The two properties can be easily proven by checking the DDT of the Keccak S-box. Suppose a 5-bit input difference of the S-box is $\delta_{in} = (\delta_{in}[4], \delta_{in}[3], \delta_{in}[2], \delta_{in}[1], \delta_{in}[0])$ and the output difference is $\delta_{out} = (\delta_{out}[4], \delta_{out}[3], \delta_{out}[2], \delta_{out}[1], \delta_{out}[0])$. Property 3 and Property 4 are then as follows.

[6] The idea of the solution set first appeared in Biham and Shamir's original work on differential cryptanalysis.

Property 3. If $\delta_{out} = 0\text{x}1$ then $\delta_{in}[0] = 1$.

Property 4. If $\delta_{out} = 0\text{x}3$ then $\delta_{in}[1] \oplus \delta_{in}[3] = 1$.

We summarise all cases with special output differences in Table 2.

Output Difference	Conditions	Output Difference	Conditions
0x1	$\delta_{in}[0] = 1$	0x3	$\delta_{in}[1] \oplus \delta_{in}[3] = 1$
0x2	$\delta_{in}[1] = 1$	0x6	$\delta_{in}[2] \oplus \delta_{in}[4] = 1$
0x4	$\delta_{in}[2] = 1$	0xc	$\delta_{in}[3] \oplus \delta_{in}[0] = 1$
0x8	$\delta_{in}[3] = 1$	0x18	$\delta_{in}[4] \oplus \delta_{in}[1] = 1$
0x10	$\delta_{in}[4] = 1$	0x11	$\delta_{in}[0] \oplus \delta_{in}[2] = 1$

Table 2. Summary of Conditions for Special Output Differences of χ.

If only one input bit of the Keccak S-box is known, two output differences of the S-box become linear. Let us show only the case when the least significant input bit is given in Property 5. The other cases are shown in Appendix A. Suppose the two 5-bit inputs of the Keccak S-box are $x_4x_3x_2x_1x_0$ and $x'_4x'_3x'_2x'_1x'_0$. The corresponding outputs are $y_4y_3y_2y_1y_0$ and $y'_4y'_3y'_2y'_1y'_0$.

Property 5. ([12]) Given x_1 and x'_1, $\delta_{out}[0]$ is a linear combination of $\delta_{in}[0]$, x_2 and x'_2. Similarly, $\delta_{out}[4]$ is a linear combination of $\delta_{in}[4]$, x_1 and x'_1.

Property 5 comes from the algebraic relation between input and output of χ. The algebraic normal form of χ is as follows:

$$\begin{aligned} y_i &= x_i + (x_{(i+1 \mod 5)} + 1) \cdot x_{(i+2 \mod 5)}, \quad &\text{for } i = 0, \ldots, 4 \\ y'_i &= x'_i + (x'_{(i+1 \mod 5)} + 1) \cdot x'_{(i+2 \mod 5)}, \quad &\text{for } i = 0, \ldots, 4 \end{aligned}$$

$\delta_{out}[0]$ and $\delta_{out}[4]$ can be written as:

$$\begin{aligned} \delta_{out}[0] &= \delta_{in}[0] + (x_1 + 1) \cdot x_2 + (x'_1 + 1) \cdot x'_2, \\ \delta_{out}[4] &= \delta_{in}[4] + (x_0 + 1) \cdot x_1 + (x'_0 + 1) \cdot x'_1. \end{aligned}$$

When x_1 and x'_1 take different values, the expressions of $\delta_{out}[0]$ and $\delta_{out}[4]$ are linearised as shown in Table 3.

Conditions	Linear Expressions	
$x_1 = x'_1 = 0$	$\delta_{out}[0] = \delta_{in}[0] + \delta_{in}[2]$	$\delta_{out}[4] = \delta_{in}[4]$
$x_1 = x'_1 = 1$	$\delta_{out}[0] = \delta_{in}[0]$	$\delta_{out}[4] = \delta_{in}[4] + \delta_{in}[0]$
$x_1 = 1,\ x'_1 = 0$	$\delta_{out}[0] = \delta_{in}[0] + x'_2$	$\delta_{out}[4] = \delta_{in}[4] + x_0 + 1$
$x_1 = 0,\ x'_1 = 1$	$\delta_{out}[0] = \delta_{in}[0] + x_2$	$\delta_{out}[4] = \delta_{in}[4] + x'_0 + 1$

Table 3. Linear Expressions of $\delta_{out}[0]$ and $\delta_{out}[4]$ with Different x_1 and x'_1

3 Guo et al.'s collision attacks on SHA-3

In this section we revisit the most dedicated existing collision attacks against the SHA-3 hash function, constructed by Guo et al. utilising an algebraic and differential hybrid method [12].

The framework of the attack against SHA3-n is shown in Figure 2. Given an n_2-round high-probability differential characteristic with the first n bits of the output difference ΔS_O as zeros, the attack consists of two stages. In the first stage the adversary applies an n_1-round connector by linearising the first n_1 rounds. Thus the adversary obtains message pairs as $\{(M_1, M_1^{'})|R_{n_1}(\overline{\mathrm{M}_1}||0) \oplus R_{n_1}(\overline{\mathrm{M}_1^{'}}||0) = \Delta S_I\}$, where ΔS_I is the input difference of the differential characteristic. In the second stage, the adversary finds a colliding pair following an n_2-round differential characteristic by searching over the pairs of messages obtained in the first stage.

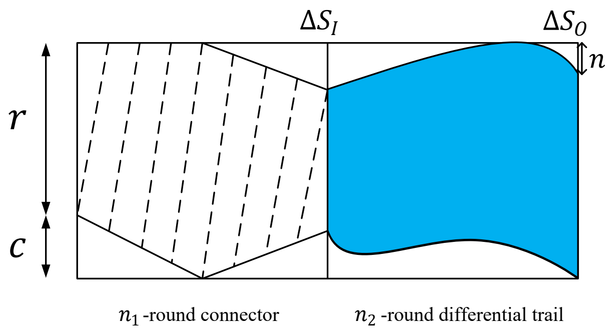

Fig. 2. Overview of Guo et al.'s $(n_1 + n_2)$-round Collision Attacks

The main drawback of this approach is that in the linearisation technique in the first stage, bit conditions are added in order to linearise the first n_1 rounds, thus consuming many degrees of freedom. As the input space of SHA3-384 is too small for a sufficient level of degrees of freedom, extra bit conditions may cause contradictions with restrictions on the initial values in the capacity part, thus making the linearisation technique infeasible.

4 New framework for a collision attack against 4-round SHA3-384

In this section we introduce a new framework for a collision attack that overcomes drawbacks in Guo et al.'s technique. There are three stages in our attack, namely the 1st block generation stage, the 1-round SAT-based connector stage, and the collision searching stage, as depicted in Figure 3. Before we overview the three stages, we introduce some notations, definitions, and parameters.

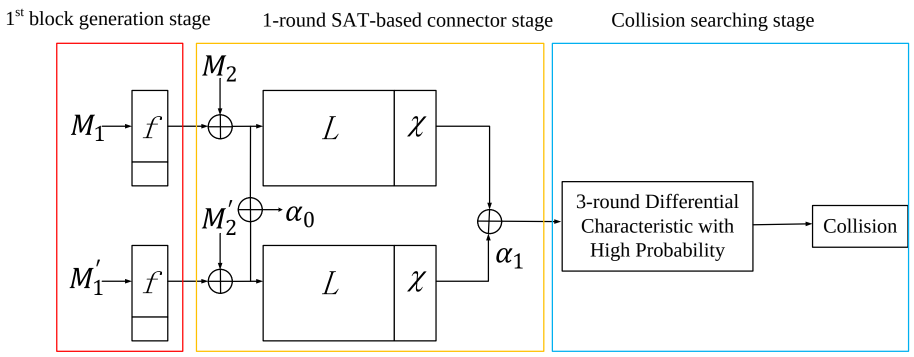
1st block generation stage
1-round SAT-based connector stage
Collision searching stage
M_1
M_1'
f
M_2
M_2'
α_0
L
χ
α_1
3-round Differential Characteristic with High Probability
Collision

Fig. 3. The framework of Our Attack

The first blocks M_1 and $M_1^{'}$ in Figure 3 are called *prefix*es. The second blocks M_2 and $M_2^{'}$ are called *suffix*es. In our attack, we assume that message pairs $(M_1||M_2)$ and $(M_1^{'}||M_2^{'})$ are actually messages after applying the padding rule. The first blocks M_1 and M_1' as well as the first 828 bits of the suffixes M_2 and $M_2^{'}$ are controlled by the adversary and the last four bits of the suffixes are 0111. The size of the input space in this case is $832 + 828 = 1660$ bits. The last $(r + 4)$ bits of the input state of the second block are defined as *chaining values*, which are known given a fixed prefix pair.

The 3-round differential characteristic depicted in Figure 3 is given in Appendix B. The probability of the characteristic is 2^{-42}. The method of constructing the characteristic is discussed in Section 5.2. The differential transition of the i-th round in the characteristic is denoted by $\alpha_{i-1} \xrightarrow{L} \beta_{i-1} \xrightarrow{\chi} \alpha_i$, where $i \geq 1$. Let α_0 be the input difference of the second block after adding message blocks as shown in Figure 3. Next, we overview the three stages.

4.1 1st block generation stage

In this stage the adversary generates prefix pairs fulfilling required conditions on corresponding chaining values. The method of deriving these conditions is given in Section 5.1.

Instead of working on single-block messages like in the previous technique, we turn to two-block messages, similar to the attacks proposed by Wang et al. against MD-like hash functions [28,29,30]. This helps to reduce the impact of restrictions from the initial values in the capacity part on the 1-round connector. The Keccak permutation on a prefix is regarded as a pseudo random number generator (PRNG). Once a corresponding 1-round connector fails, the adversary just generates another random prefix pair $(M_1, M_1^{'})$ fulfilling the required conditions over the chaining values. In comparison, in Guo et al.'s method, the adversary would have to search for a new differential characteristic with a special form, which is a hard and time-consuming process.

4.2 1-round SAT-based connector stage

We develop a new 1-round SAT-based connector that replaces Guo et al.'s linearisation technique, and removes the requirement on the size of the input space. In this stage, for each prefix pair generated in the first stage, the adversary searches for a suffix pair which connects the chaining values with a preset 1-round input difference α_1. This problem is called a *connectivity problem*, and it is defined as follows.

Definition 1. *Given a prefix pair $(M_1, M_1^{'})$, the connectivity problem is to determine if there exists a suffix pair $(M_2, M_2^{'})$ such that*

$$R(f(M_1||0) \oplus (M_2||0)) \oplus R(f(M_1^{'}||0) \oplus (M_2^{'}||0)) = \alpha_1. \qquad (2)$$

The connectivity problem can be reduced to a satisfiability (SAT) problem and solved by a SAT solver. However, solving the connectivity problem using a SAT solver for every prefix pair generated in the first stage is still time consuming. Instead, we develop a preliminary *deduce-and-sieve* algorithm that filters prefix pairs based on their differential properties.

In the deduce-and-sieve algorithm, the adversary rejects a prefix pair $(M_1, M_1^{'})$ if there exists no differential transition from α_0 to α_1, where the last $1600-828 = 772$ bits of α_0 are the sum of the chaining values. Thus, the adversary can efficiently dismiss most of prefix pairs that have no solution for corresponding connectivity problems.

Then, the adversary applies the SAT solver to solve the connectivity problems for remaining pairs. For a prefix pair $(M_1, M_1^{'})$, if there exists a suffix pair $(\hat{M}_2, \hat{M}_2^{'})$ such that Equation 2 holds, the SAT solver then returns the corresponding suffix pair, which is called a *suffix seed pair*; otherwise, the SAT solver rejects the prefix pair. The 1-round SAT-based connector stage is described in more detail in Section 6.

4.3 Collision searching stage

The method in the collision searching stage follows Guo et al.'s work: once the adversary obtains a prefix pair $(M_1, M_1^{'})$ and a suffix seed pair $(\hat{M}_2, \hat{M}_2^{'})$, the adversary aims to find a suffix pair $(M_2, M_2^{'})$ following the differential characteristic depicted in Figure 3.

All solutions for a corresponding connectivity problem form an affine subspace. Next, we explain how to derive that subspace of suffixes M_2, which also applies to deriving a subspace of suffixes $M_2^{'}$.

Given a pair of prefix and suffix seeds $(\hat{M}_2, \hat{M}_2^{'})$, the input difference of the operation χ, denoted as β_0, can be deduced by computing $L(f(M_1||0) \oplus$

$(\hat{M}_2||0)) \oplus L(f(M_1'||0) \oplus (\hat{M}_2'||0))$. Let x be a vector of bit values before χ, i.e. $x = L(f(M_1||0) \oplus (M_2||0))$. By Property 2, given β_0 and α_1, all linear equations on input affine subspaces of active S-boxes in the first round can be derived and expressed as

$$A_1 \cdot x = b_1, \tag{3}$$

where A_1 is a block-diagonal matrix in which each diagonal block together with corresponding constants in b_1 forms equations for one active S-box. Additional constraint that x needs to fulfill is that given M_1 and M_1' the chaining values are prefixed:

$$A_2 \cdot x = b_2, \tag{4}$$

where A_2 is a submatrix of L^{-1} and b_2 is the vector of those prefixed chaining values. Thus, x is in an affine subspace, which is equivalent to M_2 being in an affine subspace.

The adversary combines and solves Equation 3 and Equation 4 and obtains all solutions to the connectivity problem. The adversary then exhaustively searches for solutions of collision pairs that follow a 3-round differential characteristic.

5 Constructing a 3-round differential characteristic

In this section, we first introduce the deduction of conditions on chaining values given an input difference. Afterwards, we discuss two criteria when constructing a differential characteristic. With the differential characteristic, we explain the method of generating prefix pairs satisfying conditions on corresponding chaining values.

5.1 Requirements on chaining values

For the connectivity problem to have at least one solution (at least one pair of compatible suffixes), chaining values must follow certain conditions. We classify these conditions into two types having different forms:

- **Type-I Conditions** are of the form $\alpha_0[i_1] \oplus \alpha_0[i_2] = c$, where c is a constant and $\alpha_0[i_1]$, $\alpha_0[i_2]$ $(i_1, i_2 \geq 828)$ are the i_1th bit and the i_2th bit of α_0, respectively, and $\alpha_0[i_1]$ and $\alpha_0[i_2]$ are in the same column.
- **Type-II Conditions** are of the form $\alpha_0[j_1] \oplus \alpha_0[j_2] \oplus \alpha_0[j_3] \oplus \alpha_0[j_4] = c$, where $j_1, j_2, j_3, j_4 \geq 828$ and c is a constant. Each pair of bits $(\alpha_0[j_1], \alpha_0[j_2])$ and $(\alpha_0[j_3], \alpha_0[j_4])$ are in the same column, while $\beta_0[\sigma(j_2)]$ and $\beta_0[\sigma(j_3)]$ are in the same row.

We demonstrate how to comply with these two types of conditions in chaining values. With α_1, the adversary obtains an output difference of each S-box from α_1. There are three types of output differences from which the adversary can derive conditions on β_0. These cases are listed as follows.

Algorithm 1 Deriving Type-I Conditions

```
Input: $\alpha_1$
Output: Set of Type-I Conditions $S_A$
 1: Compute the output difference for each S-box from $\alpha_1$.
 2: $S_0 = \emptyset$, $S_1=\emptyset$, $S_A = \emptyset$.
 3: for each integer $i \in [0, 1600)$ do
 4:     $\delta_{out}$ is the output difference of the corresp. S-Box where $\alpha_1[i]$ is embedded in.
 5:     if $\delta_{out} = 0$ then
 6:         $S_0 = S_0 \cup \{i\}$.
 7:     else if $\delta_{out} \in \{0x1,0x2,0x4,0x8,0x10\}$ and $\alpha_1[i] = 1$ then
 8:         $S_1 = S_1 \cup \{i\}$.                                    ▷ Property 3
 9: for each integer $i \in [828, 1600)$ do
10:     for each integer $j \in [i + 1, 1600)$ do
11:         if $\sigma(i), \sigma(j) \in S_0$ or $\sigma(i), \sigma(j) \in S_1$ then
12:             $S_A = S_A \cup \{\alpha_0[i] \oplus \alpha_0[j] = 0\}$.          ▷ Property 1
13:         else if $\sigma(i) \in S_0, \sigma(j) \in S_1$ or $\sigma(i) \in S_1, \sigma(j) \in S_0$ then
14:             $S_A = S_A \cup \{\alpha_0[i] \oplus \alpha_0[j] = 1\}$.          ▷ Property 1
15: Remove linear related conditions in $S_A$
16: return $S_A$
```

- **Type-I Output Difference**: The output difference of an S-box is zero when it is inactive.
- **Type-II Output Difference**: The output difference of an S-box is 0x1, 0x2, 0x4, 0x8, or 0x10.
- **Type-III Output Difference**: The output difference of an S-box is 0x3, 0x6, 0xc, 0x11 or 0x18.

As from the form of Type-I Conditions, the adversary can derive conditions from Type-I and Type-II output differences by applying CP-kernel equations. The procedure is shown in Algorithm 1. From Line 3 to Line 8, the adversary obtains Type-I and Type-II Output Differences from α_1. Then, the adversary deduces Type-I Conditions on chaining values by applying Property 1.

With these three types of output differences, the adversary can derive Type-II Conditions. The procedure is shown in Algorithm 2. For each Type-III Output Difference δ_{out}, the adversary checks Table 2 and writes a corresponding condition as $\beta_0[i_1] + \beta_0[i_2] = 1$. If both of the bits $\alpha_0[\sigma^{-1}(i_1)]$ and $\alpha_0[\sigma^{-1}(i_2)]$ are in the chaining value part, the procedure continues; otherwise, the adversary moves to the next S-box. As shown in Lines 8 to 22 in Algorithm 2, if $\alpha_0[\sigma^{-1}(i_1)]$ and $\alpha_0[\sigma^{-1}(i_2)]$ are in the chaining value part, the adversary checks CP-kernel equations where $\beta_0[i_1]$ and $\beta_0[i_2]$ are involved in:

$$\alpha_0[\sigma^{-1}(i_1)] \oplus \alpha_0[j_1] = \beta_0[i_1] \oplus \beta_0[\sigma(j_1)],$$
$$\alpha_0[\sigma^{-1}(i_2)] \oplus \alpha_0[j_2] = \beta_0[i_2] \oplus \beta_0[\sigma(j_2)],$$

where $\alpha_0[j_1]$ and $\alpha_0[j_2]$ are in the same column with $\alpha_0[\sigma^{-1}(i_1)]$ and $\alpha_0[\sigma^{-1}(i_2)]$, respectively. The adversary searches for $\alpha_0[j_1]$ and $\alpha_0[j_2]$ with some extra requirements: $\alpha_0[j_1]$ and $\alpha_0[j_2]$ should be in the chaining value part; $\beta_0[\sigma(j_1)]$ and

Algorithm 2 Deriving Type-II Conditions

```
Input: $\alpha_1$
Output: Set of Type-II Conditions $S_B$
1: Compute the output difference for each S-box from $\alpha_1$.
2: $S_0$, $S_1$ are the same as in Algorithm 1, $S_B = \emptyset$.
3: for each S-box do
4:     $\delta_{out}$ is the output difference of the S-Box.
5:     if $\delta_{out} \in \{0x3,0x6,0xc,0x11,0x18\}$ then
6:         flag= 0.
7:         Check Table 2 and write the corresponding condition as $\beta_0[i_1] + \beta_0[i_2] = 1$.
8:         if $\sigma^{-1}(i_1) \geq 828$ and $\sigma^{-1}(i_2) \geq 828$ then
9:             for each $\alpha_0[j_1]$ in the $\psi_0[\sigma^{-1}(i_1)]$th column where $j_1 \geq 828$ do
10:                if $\sigma(j_1) \in S_0$ or $\sigma(j_1) \in S_1$ then
11:                    flag=1.
12:                    $c_1 = i$, where $i$ is 0/1 s.t. $\sigma(j_1) \in S_i$
13:                    Break
14:            if (flag) then
15:                flag=0.
16:                for each $\alpha_0[j_2]$ in the $\psi_0[\sigma^{-1}(i_2)]$th column where $j_2 \geq 828$ do
17:                    if $\sigma(j_2) \in S_0$ or $\sigma(j_2) \in S_1$ then
18:                        flag=1.
19:                        $c_2 = i$, where $i$ is 0/1 s.t. $\sigma(j_2) \in S_i$
20:                        Break
21:                if (flag) then
22:                    $S_B = S_B \cup \{\alpha_0[j_1] \oplus \alpha_0[\sigma^{-1}(i_1)] \oplus \alpha_0[\sigma^{-1}(i_2)] \oplus \alpha_0[j_2] = 1 \oplus c_1 \oplus c_2\}$
23: return $S_B$
```

$\beta_0[\sigma(j_2)]$ should be embedded in S-boxes with either Type-I or Type-II Output Differences. If such $\alpha_0[j_1]$ and $\alpha_0[j_2]$ exist, then the adversary derives a Type-II Condition (Line 22 in Algorithm 2).

5.2 How to construct a 3-round differential characteristic

The 3-round differential characteristic in our attack adapts the second characteristic in [12, Table 9]. The last two rounds of their characteristic are used as the last two rounds characteristic from the third round to the fourth round in our attack. We slightly change the output difference of their characteristic to make the first 384 bit differences be zero. Thus, the probability of the last two rounds characteristic is 2^{-16} instead of 2^{-15} in the original one.

We extend the 2-round backward characteristic by one extra round. When β_1 is fixed, the 3-round differential characteristic is determined. We choose β_1 according to two criteria as follows:

- **Criterion 1**: The affine subspace in the collision searching stage should be sufficiently large to find a collision pair.
- **Criterion 2**: The number of conditions on the chaining values should not be too large.

If Criterion 1 is not fulfilled, the affine subspace defined by Equation 3 and Equation 4 is so small that the probability that a collision pair is obtained in the third stage becomes insignificant.

If the characteristic does not follow Criterion 2, the procedure of generating 1st message blocks will become infeasible to be realised in practice. To keep our attack practical, the differential characteristic should satisfy Criterion 2.

The difference α_2 has 8 active S-boxes in the second round. From the DDT of the S-box, the probability of a nonzero differential transition is at least 2^{-4}. Thus, the probability of our 3-round differential characteristic is not larger than $(2^{-4})^8 \cdot 2^{-16} = 2^{-48}$. The size of the affine subspace in the collision searching stage should be larger than 48. The probability of the first round transition should not be smaller than $2^{828-48} = 2^{780}$. As the average probability of a nonzero differential path is 2^{-3}, there should be no more than $780/3 = 260$ active S-boxes in the first round to satisfy Criterion 1.

As for Criterion 2, we set the threshold for the number of conditions as 50. We use a hash table to generate prefix pairs, as discussed in Section 5.3. When the number of conditions is too large, the memory consumption when generating 1st blocks is infeasible.

We use a greedy algorithm to choose β_1. With α_2 from the second characteristic in [12, Table 9], the adversary picks a compatible β_1 at random and computes α_1 as $L^{-1}(\beta_1)$. Given α_1, the adversary obtains the number of active S-boxes in the first round. With Algorithm 1 and Algorithm 2, the adversary deduces conditions on the chaining values. If the two criteria are satisfied, the greedy algorithm outputs a 3-round characteristic; otherwise, the adversary pick another compatible β_1 and continue the procedure.

In our differential characteristic presented in Table 6, there are 228 active S-boxes in the first round. Applying Algorithm 1 and Algorithm 2 on the differential characteristic, there are 39 conditions on the chaining values, including 38 Type-I Conditions and 1 Type-II Condition. These conditions are listed in Appendix C. The probability of the differential characteristic is 2^{-42}. Our differential characteristic may not minimise the attack's complexity. Finding such a differential characteristic is an open problem.

5.3 Generating prefix pairs fulfilling the requirements

We explain the generation procedure of prefix pairs fulfilling the 39 conditions in Table 7. In our approach, we use a hash table to trade off memory for time and data complexities. The memory consumption in this procedure is mainly from a hash table indexed by 39 bits. Before we describe the procedure, we introduce some definitions.

We define a *constant sum* as a binary string $c_{38}c_{37}\cdots c_1c_0$, where c_i is the sum in the i-th condition, $0 \leq i \leq 38$. In our case, the constant sum is 0x7e00000000 from the conditions in Table 7. The conditions are the sums of differences in certain bit positions of the input state of the second block. To check whether given a prefix pair $(M_1, M_1^{'})$ the conditions are satisfied, the adversary first computes sums of binary values in these positions, which are called the *value*

Algorithm 3 Generating Prefix Pairs

```
Output: Set of Prefix Pairs $S_P$
 1: Constant sum $\Sigma$=0x7e00000000
 2: $S_P = \emptyset$
 3: Initialise an array Counter of length $2^{39}$ with zeros.
 4: for each integer $i \in [0, 2^n)$ do
 5:     Randomly pick a message $M$ of 832 bits and compute the value string $c$.
 6:     HashTable[$c$][Counter[$c$]]=M
 7:     Increase Counter[$c$] by 1.
 8: for each integer $i \in [0, 2^n)$ do
 9:     if $i < i \oplus \Sigma$ then
10:         for each integer $j \in [0, \text{Counter}[i])$ do
11:             for each integer $k \in [0, \text{Counter}[i \oplus \Sigma]]$ do
12:                 $S_P = S_P \cup \{(\text{HashTable}[i][j], \text{HashTable}[i \oplus \Sigma][k])\}$
```

string and recorded as a 39-bit string. Then, the adversary sums up value strings and checks whether the result equals the constant sum value. If true, then the adversary obtains a prefix pair fulfilling all conditions; otherwise discards it.

The procedure of generating prefix pairs satisfying the conditions is shown in Algorithm 3. First, the adversary generates 2^n messages M of length 832 bits and computes corresponding value strings c. The adversary places M into the cth row of the hash table. Thereafter, the adversary searches through the hash table for prefix pairs that satisfy the constraints (Lines 8 to 16).

Algorithm 3 generates around $2^n \cdot 2^{n-1} \cdot 2^{-39} = 2^{2n-40}$ pairs. The time and data complexity is 2^n. The memory consumption is mainly from the hash table, which is also 2^n. The value of n is experimentally discussed in Section 7.

6 1-round SAT-based connector

We develop a new 1-round SAT-based connector to solve the connectivity problem in an efficient way. The connector includes two phases. First, we use a deduce-and-sieve algorithm to filter prefix pairs generated by Algorithm 3. Then, for each remaining prefix pair, the connectivity problem is resolved by applying a SAT solver.

6.1 Deduce-and-sieve algorithm

In the deduce-and-sieve algorithm, we assume that for a prefix pair $(M_1, M_1^{'})$, there exists a suffix pair $(M_2, M_2^{'})$ in the connectivity problem. It indicates that in some S-boxes, input differences for Type-I and Type-II output differences should be of a special form. Thus, some bit differences of β_0 are supposed to be fixed, which are then recorded in the sets S_0 and S_1, see Algorithm 1.

There are two phases in the deduce-and-sieve algorithm. In the *difference phase*, given a prefix pair $(M_1, M_1^{'})$ the adversary deduces new bit differences and checks whether a contradiction has been reached, in which case the prefix

Algorithm 4 Initial Phase of the Deduce-and-sieve Algorithm

```
1: procedure INITIAL($M_1$, $M_1^{'}$)
2:     $A = f(M_1||0)$, $A' = f(M_1^{'}||0)$
3:     for each integer $i \in [0, 1600)$ do
4:         if $i \geq 828$ then
5:             $\alpha_0[i] = A[i] \oplus A'[i]$,   $\alpha_0^S[i] = 1$,   $A_S[i] = 1$,   $A'_S[i] = 1$
6:         else
7:             $\alpha_0[i] = 0$,   $\alpha_0^S[i] = 0$,   $A[i] = 0$, $A'[i] = 0$,   $A_S[i] = 0$,   $A'_S[i] = 0$
8:         if $i \in S_0$ then
9:             $\beta_0[i] = 0$,   $\beta_0^S[i] = 1$
10:        else if $i \in S_1$ then
11:            $\beta_0[i] = 1$,   $\beta_0^S[i] = 1$
12:        else
13:            $\beta_0^S[i] = 0$
14:        $B[i] = 0$,   $B'[i] = 0$,   $B_S[i] = 0$,   $B'_S[i] = 0$
15:    for each integer $i \in [0, 320)$ do
16:        $\Sigma[i] = 0$,   $\Sigma^S[i] = 0$
17:    return $A, A', A_S, A'_S, B, B', B_S, B'_S, \alpha_0, \alpha_0^S, \beta_0, \beta_0^S$
```

pair is discarded. The *value phase* helps to sieve prefix pairs more efficiently, as the filtering rate of the difference phase is low. In the value phase, more bit values can be deduced from the algebraic property of the Keccak S-box. If new bit differences are obtained from new bit values, the adversary returns to the difference phase to seek a contradiction and to discard the prefix pair.

In the initial phase of the deduce-and-sieve algorithm (Algorithm 4), the adversary computes the chaining values for a prefix pair $(M_1, M_1^{'})$ and stores the values in two vectors A and A' of length 1600, respectively. As the bit values in vectors A and A' can be either known or unknown, two extra vectors A_S, A'_S, called *indicator vectors*, record whether the bit value in a corresponding position is known. To be more specific, for $0 \leq i < 1600$, if and only if the ith bit in A is known then $A_S[i] = 1$. The adversary then computes the bit difference $\alpha_0[i]$, where $828 \leq i < 1600$, and sets the corresponding bit differences of β_0 as a constant vector. Two indicator vectors α_0^S and β_0^S record whether the bit difference in a corresponding position is known.

In the deduce-and-sieve algorithm, we use the vector Σ of length 320 to record sums of five bit differences in each column. In the beginning, each entry of the indicator vector Σ^S is initialised as 0 denoting an unknown state.

6.1.1 The difference phase

In the difference phase of the deduce-and-sieve algorithm, for a given prefix pair $(M_1, M_1^{'})$, bit differences of α_0 in the chaining value part can be obtained. As we assume that there exists a solution in the connectivity problem for the prefix pair $(M_1, M_1^{'})$, some bit differences of β_0 should be certain values from the sets S_0 and S_1, deduced by Algorithm 1.

With the CP-kernel equations and the expression of L, the adversary can deduce new bit differences from α_0 and β_0. We investigate the differential transition of the Keccak S-box and develop a tool called *Truncated Difference Transform Table* (TDTT), which is inspired by the truncated differential cryptanalysis, proposed by Knudsen in [15]. With the TDTT of the Keccak S-box, a contradiction may be reached for the prefix pair $(M_1, M_1^{'})$ as there is no compatible differential path from the input difference of the first round α_0 to the output difference α_1. Before we define the TDTT of an S-box, we first introduce truncated difference in Definition 2.

Definition 2. *An n-bit difference is called a* truncated difference *if only m bits of it are known, where $m < n$.*

A $2n$-bit integer $a||b$, where $a, b \in \mathbb{F}_2^n$, is used to represent a truncated difference. Let $(a_{n-1}, \cdots, a_0)$ and $(b_{n-1}, \cdots, b_0)$ be the binary representations of a and b, respectively. For each i where $0 \leq i < n$, if b_i is known, $a_i = 1$; otherwise, $a_i = 0$. In *regular* truncated differences, for each i where $0 \leq i < n$, $a_i \geq b_i$. Otherwise, the truncated difference is called *irregular*.

Differences *covered* by a regular truncated difference $a||b$, where $a, b \in \mathbb{F}_2^n$, are $\{d \in \mathbb{F}_2^n | d_i = b_{n+i}$ if $a_i = 1, 0 \leq i < n\}$. A difference $d \in \mathbb{F}_2^n$ being covered by a truncated difference $\Delta \in \mathbb{F}_2^{2n}$ is denoted by $d \preceq \Delta$.

We observe that with an output difference of the Keccak S-box and a corresponding truncated input difference, more bits of the input difference can be fixed. For example, suppose that the output difference of an S-box is 0x1 and the truncated input difference is 0 where none of the input bit differences is known. Property 3 indicates that the least significant input bit difference should be 1. Then, the truncated input difference of the S-box can be updated as 0000100001 in the binary representation. A complete transforming behaviour of the S-box can be described by its TDTT:

Definition 3. *Given a truncated input difference Δ_{in}^T and an output difference Δ_{out}, the entry $TDTT(\Delta_{in}^T, \Delta_{out})$ of the S-box's TDTT is:*

$$TDTT(\Delta_{in}^T, \Delta_{out}) = \begin{cases} null, & \text{if } \Delta_{in}^T \text{ does not deduce } \Delta_{out}, \text{ or } \Delta_{in}^T \text{ is irregular} \\ \Delta_{in}^{T'}, & \text{if more bits of the input difference can be derived} \\ \Delta_{in}^T, & \text{if no more bits can be derived} \end{cases}$$

where $\Delta_{in}^{T'}$ is the new truncated input difference, $\Delta_{in}^T, \Delta_{in}^{T'} \in \mathbb{F}_2^{2n}$ and $\Delta_{out} \in \mathbb{F}_2^n$.

In case of Property 3, it can be observed that TDTT$(0, 1)$ is 0x21.

The TDTT of an S-box can be constructed from its DDT, which is shown in Algorithm 5. For a regular truncated difference $a \in \mathbb{F}_2^{2n}$ and an output difference $b \in \mathbb{F}_2^n$, the adversary finds all the covered differences Δ_{in} by a such that the differential path $\Delta_{in} \rightarrow b$ is compatible, where $\Delta_{in} \in \mathbb{F}_2^n$ (Line 4 in Algorithm 5). If there exists no such Δ_{in}, TDTT(a, b) =null. Otherwise, the adversary finds the new truncated difference T covering all Δ_{in} and $\mathrm{TDTT}(a, b) = T$ (Line 8 to 14 in Algorithm 5). For an irregular truncated difference $a \in \mathbb{F}_2^{2n}$, the adversary labels the entire row of the TDTT as null, shown in Line 16 of Algorithm 5.

Algorithm 5 Constructing the TDTT of an n-bit S-box from its DDT

```
Input: DDT of an $n$-bit S-box
Output: TDTT of the S-box
 1: for each integer $a \in [0, 2^{2n})$ do
 2:     if $a$ is a regular truncated difference then
 3:         for each integer $b \in [0, 2^n)$ do        ▷ TDDT$(a, b)$ is computed in the loop.
 4:             Find $D = \{\Delta_{in} \in \mathbb{F}_2^n | \mathrm{DDT}(\Delta_{in}, b) \neq 0\} \cap \{\Delta_{in} \in \mathbb{F}_2^n | \Delta_{in} \preceq a\}$
 5:             if $D = \emptyset$ then
 6:                 TDTT($a$,$b$)=null
 7:             else
 8:                 $T$=0                                ▷ $T$ is a $2n$-bit integer.
 9:                 for each integer $i \in [0, n)$ do
10:                     if the $i$-th bit of each entry in $D$ is a constant as $c_i$ then
11:                         $T_i = c_i, T_{n+i} = 1$          ▷ $T_i$ is the $i$th bit of $T$.
12:                     else
13:                         $T_i = 0, T_{n+i} = 0$
14:                 $\mathrm{TDTT}(a, b) = T$
15:     else
16:         TDTT($a$,*)=null
17: return TDTT
```

Next, we describe the method of deducing new bit differences from the CP-kernel equations and the expression of L. Then, we explain the method of applying it for sieving prefix pairs.

Deducing New Differences from the CP-kernel equations. New differences of some bit positions can be derived from the CP-kernel equations. For example, as shown in Figure 4, the differences $\alpha[i_3]$, $\alpha[i_4]$ and $\beta[\sigma(i_3)]$ are known. $\beta[\sigma(i_4)]$ can be deduced from the CP-kernel equation as $\beta[\sigma(i_4)] = \alpha[i_3] \oplus \alpha[i_4] \oplus \beta[\sigma(i_3)]$. New

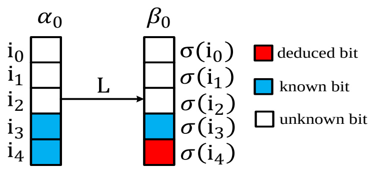

Fig. 4. Deducing New Differences from the CP-kernel Equations

differences can be derived in a column if there exists a position i_j in the column such that both $\alpha[i_j]$ and $\beta[\sigma(i_j)]$ are known, and $0 \leq j < 5$. The procedure of deducing new differences in the i-th Column is shown in Algorithm 6.

Algorithm 6 Deducing New Differences in the i-th Column

```
 1: procedure CPKernel($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $i$)
 2:     Compute the indices of the five bits in the $i$-th column as $i_0, i_1, \cdots, i_4$.
 3:     $flag = 0$
 4:     for each integer $j \in [0, 5)$ do
 5:         if $\alpha_0^S[i_j] = 1$ and $\beta_0^S[\sigma(i_j)] = 1$ then
 6:             $flag = 1$,  $sum = \alpha_0[i_j] \oplus \beta_0[\sigma(i_j)]$
 7:     if $flag$ then
 8:         for each integer $j \in [0, 5)$ do
 9:             if $\alpha_0^S[i_j] = 1$ and $\beta_0^S[\sigma(i_j)] = 0$ then
10:                 $\beta_0[\sigma(i_j)] = sum \oplus \alpha_0[i_j]$,  $\beta_0^S[\sigma(i_j)] = 1$.
11:             else if $\alpha_0^S[i_j] = 0$ and $\beta_0^S[\sigma(i_j)] = 1$ then
12:                 $\alpha_0[i_j] = sum \oplus \beta_0[\sigma(i_j)]$,  $\beta_0^S[\sigma(i_j)] = 1$
```

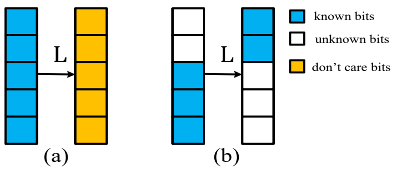

Fig. 5. Representatives when the Sum State is 1

Deducing New Differences from the Expression of L. New bit differences can be derived from the expression of L. Applying Equation 1, the bit difference $\beta_0[i]$ can be expressed as:

$$\beta_0[i] = \alpha_0[\sigma^{-1}(i)] \oplus \Sigma[\phi_1(\sigma^{-1}(i))] \oplus \Sigma[\phi_2(\sigma^{-1}(i))], \tag{5}$$

where $\Sigma[\phi_1(\sigma^{-1}(i))]$ and $\Sigma[\phi_2(\sigma^{-1}(i))]$ are the sums of the five bits in the $\phi_1(\sigma^{-1}(i))$th and $\phi_2(\sigma^{-1}(i))$th columns, respectively, and $0 \leq i < 1600$. If only one variable in Equation 5 is unknown, its value can be deduced. Before we show the technique of deducing new difference applying Equation 5, we introduce the method of computing the column sum.

We classify the situations of the column sum into three cases. The first case is that the column sum is known. The second is that the column sum becomes known with one more known bit. The third is that more than two bits of information are needed to derive the column sum. In order to compute the column sum, we use another variable called a *sum state* that encodes the three states: 1 for the first case, 2 for the second case, or 0 otherwise.

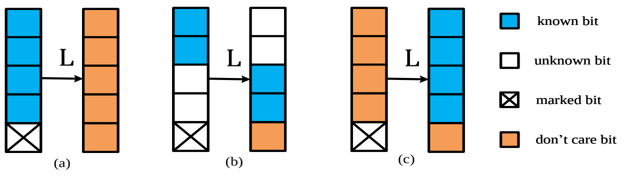

Fig. 6. Representatives when the Sum State is 2

Representatives when the sum state is 1 are shown in Figure 5.[7] For example, as shown in Figure 5 (b), $\alpha_0[i_2]$, $\alpha_0[i_3]$, $\alpha_0[i_4]$, $\beta_0[\sigma(i_0)]$ and $\beta_0[\sigma(i_1)]$ are known. Applying the CP-kernel equation, the column sum can be computed as

$$\alpha_0[i_2] \oplus \alpha_0[i_3] \oplus \alpha_0[i_4] \oplus \beta_0[\sigma(i_0)] \oplus \beta_0[\sigma(i_1)].$$

Representatives when the sum state is 2 are shown in Figure 6.[8] In these five representatives the sum of four bits of one column can be derived with the CP-kernel equation. The index of the left bit in the column, defined as the *marked bit*, is recorded. The difference of the marked bit may be deduced in a later step.

The procedure for computing the i-th column sum is shown in Algorithm 7. The adversary finds the states of the 10 related bits from α_0 and β_0. If the situation matches one of the representatives in Figure 5, the column sum is known. The adversary computes the sum and updates the sum state $\Sigma^S[i]$ with 1. If the situation matches one of the representatives in Figure 6, where one bit difference is missing, the adversary computes the sum of the corresponding blue bits. The adversary then records the sum and the index of the marked bit and updates the sum state $\Sigma^S[i]$ with 2. If the situation does not match any of the representatives in Figure 5 or Figure 6, the adversary labels the corresponding sum state as 0.

Given column sums, the adversary can obtain new bit differences from the expression of θ. To be more specific, if there is only one variable is unknown in Equation 5, the value of it can be easily deduced. The procedure is shown in Algorithm 8. There are 6 situations in which new differences can be deduced. In the forth and sixth situations, marked bits are obtained and corresponding column sums are updated. In the first, second, third and fifth situations, when new values in α_0 or β_0 are obtained, the corresponding CP-kernel equations are checked again to deduce new bit differences.

Sieving Prefix Pairs with TDTT. The sieving procedure applying the TDTT of the Keccak S-box is shown in Algorithm 9. With β_0 and β_0^S, the adversary

[7] We only show a representative in Figure 5 (b). The other cases can be obtained by permuting the bits in the two columns simultaneously.

[8] We only show three representatives in Figure 6. The other cases can be obtained by permuting the bits in the two columns of the subfigures simultaneously.

Algorithm 7 Computing the i-th Column Sum

```
1: procedure ColumnSum($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $i$, $\Sigma$, $\Sigma^S$, $MarkedBit$)
2:     Compute the indices of the five bits in the $i$-th column as $i_0, i_1, \cdots, i_4$.
3:     if the five bits match one of the representatives in Figure 5 then
4:         Compute the sum and record it in $\Sigma[i]$
5:         $\Sigma^S[i] = 1$
6:     else if the five bits match one of the representatives in Figure 6 then
7:         Compute the sum of the four corresponding known bits and record it in $\Sigma[i]$
8:         $\Sigma^S[i] = 2$
9:         Record the index of the corresponding marked bit in $MarkedBit[i]$
10:    else
11:        $\Sigma^S[i] = 0$
```

can deduce the truncated input difference for each S-box. Then, the adversary discards prefix pairs with no solutions in the connectivity problem according to the TDTT. For pairs that cannot be discarded, the adversary may obtain new bit differences of β_0 from the TDTT. Once a bit difference is obtained, the adversary checks the related CP-kernel equations to deduce new bit differences in α_0 and β_0, as shown in Lines 13-15 in Algorithm 9.

Let us summarise the difference phase of the derive-and-seive algorithm in Algorithm 10. The adversary first initialises the difference phase by deducing bit differences through checking the CP-kernel equations. Then, she updates column sums and deduces new bit differences from the expression of L. Finally, the adversary checks the TDTT of the Keccak S-box and decides whether a certain prefix pair $(M_1, M_1^{'})$ should be discarded. If the pair should not be discarded, the adversary computes the number of new deduced bit differences from Lines 8-13 in Algorithm 10. If new bit differences are deduced, the adversary goes back to Line 8; otherwise, she accepts the prefix pair.

6.1.2 The value phase

In the value phase, the adversary applies an algebraic property of the Keccak round function to deduce new input bit values of χ in the first round. New values can be deduced using a new tool called *Fixed Value Distribution Table* (FVDT) and applying the CP-kernal Equalities.

The FVDT is developed from an observation that with a truncated input difference and an output difference of the Keccak S-box, bit values in some positions are constants. Applying the FVDT of the Keccak S-box, the adversary can obtain new input bit values of χ. With these new values and the chaining values, the adversary applies the CP-kernal equations to deduce additional input bits of χ. With the new values, the adversary can derive new bit differences from the expression of χ. Then, she can continue with the difference phase.

Fixed Value Distribution Table (FVDT). Before we delve into details of the FVDT of an n-bit S-box, we first define a solution set of a truncated input difference Δ_{in}^T and an output difference Δ_{out} as follows:

Algorithm 8 Deducing New Differences from the Expression of L

```
1: procedure LinearTrans($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\Sigma$, $\Sigma^S$, $MarkedBit$)
2:     for each integer $i \in [0, 1600)$ do
3:         $i_0 = \sigma^{-1}(i)$,  $i_1 = \phi_1(i_0)$,  $i_2 = \phi_2(i_0)$
4:         Deduce the expression of $\beta_0[i] = \alpha_0[i_0] \oplus \Sigma[i_1] \oplus \Sigma[i_2]$
5:         if $\beta_0^S[i] = 0$ and $\alpha_0^S[i_0] = 1$ and $\Sigma^S[i_1] = 1$ and $\Sigma^S[i_2] = 1$ then
6:             $\beta_0[i] = \alpha_0[i_0] \oplus \Sigma[i_1] \oplus \Sigma[i_2]$,  $\beta_0^S[i] = 1$
7:             CPkernel($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\phi_0(i_0)$)
8:         else if $\beta_0^S[i] = 1$ and $\alpha_0^S[i_0] = 0$ and $\Sigma^S[i_1] = 1$ and $\Sigma^S[i_2] = 1$ then
9:             $\alpha_0[i_0] = \beta_0[i] \oplus \Sigma[i_1] \oplus \Sigma[i_2]$,  $\alpha_0^S[i_0] = 1$
10:            CPkernel($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\phi_0(i_0)$)
11:        else if $\beta_0^S[i] = 1$ and $\alpha_0^S[i_0] = 1$ and $\Sigma^S[i_1] = 0$ and $\Sigma^S[i_2] = 1$ then
12:            $\Sigma[i_1] = \beta_0[i] \oplus \alpha_0[i_0] \oplus \Sigma[i_2]$,  $\Sigma^S[i_1] = 1$
13:        else if $\beta_0^S[i] = 1$ and $\alpha_0^S[i_0] = 1$ and $\Sigma^S[i_1] = 2$ and $\Sigma^S[i_2] = 1$ then
14:            $\alpha_0[MarkedBit[i_1]] = \beta_0[i] \oplus \alpha_0[i_0] \oplus \Sigma[i_1] \oplus \Sigma[i_2]$
15:            $\alpha_0^S[MarkedBit[i_1]] = 1$
16:            $\Sigma[i_1] = \alpha_0[MarkedBit[i_1]] \oplus \Sigma[i_1]$,  $\Sigma^S[i_1] = 1$
17:            CPkernel($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\phi_0(MarkedBit[i_1])$)
18:        else if $\beta_0^S[i] = 1$ and $\alpha_0^S[i_0] = 1$ and $\Sigma^S[i_1] = 1$ and $\Sigma^S[i_2] = 0$ then
19:            $\Sigma[i_2] = \beta_0[i] \oplus \alpha_0[i_0] \oplus \Sigma[i_1]$,  $\Sigma^S[i_2] = 1$
20:        else if $\beta_0^S[i] = 1$ and $\alpha_0^S[i_0] = 1$ and $\Sigma^S[i_1] = 1$ and $\Sigma^S[i_2] = 2$ then
21:            $\alpha_0[MarkedBit[i_2]] = \beta_0[i] \oplus \alpha_0[i_0] \oplus \Sigma[i_1] \oplus \Sigma[i_2]$
22:            $\alpha_0^S[MarkedBit[i_2]] = 1$
23:            $\Sigma[i_2] = \alpha_0[MarkedBit[i_2]] \oplus \Sigma[i_2]$,  $\Sigma^S[i_2] = 1$
24:            CPkernel($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\phi_0(MarkedBit[i_2])$)
```

Definition 4. *The solution set of a truncated input difference Δ_{in}^T and an output difference Δ_{out} is*

$$S_T(\Delta_{in}^T, \Delta_{out}) = \bigcup_{\Delta_{in} \preceq \Delta_{in}^T} \{(a, a \oplus \Delta_{in}) | S(a) \oplus S(a \oplus \Delta_{in}) = \Delta_{out}\},$$

where $\Delta_{in}^T \in \mathbb{F}_2^{2n}$, $\Delta_{in} \in \mathbb{F}_2^n$ and $\Delta_{out} \in \mathbb{F}_2^n$.

The solution set of the truncated input difference Δ_{in}^T and the output difference Δ_{out} is a generalisation of the solution set of the input difference Δ_{in} and the output difference Δ_{out} of regular DDTs. From Definition 4, it is a union of solution sets of the input difference Δ_{in} and the output difference Δ_{out}, where $\Delta_{in} \preceq \Delta_{in}^T$.

We observe that for the truncated input difference Δ_{in}^T and the output difference Δ_{out}, some bits of the pairs in the solution set $S_T(\Delta_{in}^T, \Delta_{out})$ may be constants. For example, when the truncated input difference Δ_{in}^T is 0xc2, the covered differences are 0x2, 0x3, 0xa, 0xb, 0x12, 0x13, 0x1a, and 0x1b. If the output difference Δ_{out} is 0x1, the compatible differences are 0xb and 0x1b. The solution set is S_T(0xc2,0x1) = {0x0, 0x3, 0x8, 0xb} ∪ {0x1, 0x1c}. It can be easily verified that the value of the third bit in the solution set is fixed as 0.

Based on this observation, we develop a useful tool called *Fixed Value Distribution Table.* If the adversary can obtain fixed values in some bit positions

Algorithm 9 Discarding Prefix Pairs with TDTT

```
1: procedure SIEVE($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, TDTT)
2:     for each S-box do
3:         Deduce the output difference $\Delta_{out}$ from $\alpha_1$.
4:         Deduce the truncated input difference $\Delta_{in}^T$ from $\beta_0$ and $\beta_0^S$.
5:         $T$=TDTT($\Delta_{in}^T$,$\Delta_{out}$)
6:         if $T$=null then
7:             return 0.
8:         else if $T$=$\Delta_{in}^T$ then
9:             continue
10:        else if $T \neq \Delta_{in}^T$ then
11:            Find the indices of the five bits in the S-box as $i_0, i_1, \cdots, i_4$.
12:            for each integer $j \in [0, 5)$ do
13:                if the $(j+5)$th bit of $\Delta_{in}$ is 0 and $T_{j+5} = 1$ then
14:                    $\beta_0[i_j] = T_j$, $\beta_0^S[i_j] = 1$          ▷ $T_j$ is the $j$th bit of $T$.
15:                    CPKERNEL($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\phi_0(\sigma^{-1}(i_j))$)
```

with Δ_{in}^T and Δ_{out}, we use a $2n$-bit integer $a||b$, called as fixed point, to record constant values and their corresponding positions, where $a, b \in \mathbb{F}_2^n$. If the i-th bit in the S-box is a constant, $a_i = 1$ and b_i is assigned to be a fixed value; otherwise, $a_i = 0$ and $b_i = 0$, where $0 \leq i < n$ and a_i and b_i are the i-th bit of a and b, respectively. We define the FVDT of an S-box as follows:

Definition 5. *Given a truncated input difference Δ_{in}^T and an output difference Δ_{out}, the entry $FVDT(\Delta_{in}^T, \Delta_{out})$ of the S-box's FVDT is:*

$$FVDT(\Delta_{in}^T, \Delta_{out}) = \begin{cases} null, & \text{if } \Delta_{in}^T \text{ does not deduce } \Delta_{out}, \text{ or } \Delta_{in}^T \text{ is irregular.} \\ v, & \text{otherwise.} \end{cases}$$

where Δ_{in}^T, $v \in \mathbb{F}_2^{2n}$, $\Delta_{out} \in \mathbb{F}_2^n$ and v is the fixed point with respect to Δ_{in}^T and Δ_{out}.

The adversary uses the FVDT of the Keccak S-box to initialise the value phase. The process is shown in Algorithm 11.

Deducing New Values from CP-kernel equations. Similarly to the difference phase, new values can be deduced from the CP-kernel equations. For each column, the adversary just calls CPKERNEL(A, B, A_S, B_S, i) and CPKERNEL(A', B', A'_S, B'_S, i) corresponding to two prefixes M_1 and M'_1, where i is the index of the column and $0 \leq i < 320$.

It should be noted that more operations, including the column sum technique in the differential phase (even on a fraction of columns), can be applied similarly to deduce more bit values in the value phase. The technique might help to improve the filtering rate of the deduce-and-sieve algorithm but increase its complexity. It is, however, an open problem how to balance the complexity and filtering rate of the deduce-and-sieve algorithm.

Algorithm 10 Difference Phase of the Derive-and-seive Algorithm

```
 1: procedure InitialiseDP($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$)
 2:     for each integer $i \in [0, 320)$ do
 3:         CPkernel($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $i$)
 4:     return $a$ =the number of new deduced differences bits in $\alpha_0$ and $\beta_0$
 5: procedure DP($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, TDTT)
 6:     $a$ =InitialiseDP($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$)
 7:     while $a \neq 0$ do
 8:         for each integer $i \in [0, 320)$ do
 9:             if $\Sigma^S[i] = 0$ then
10:                 ColumnSum($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $i$, $\Sigma$, $\Sigma^S$ $MarkedBit$)
11:             else if $\Sigma^S[i] = 2$ and $\alpha_0^S[MarkedBit[i]] = 1$ then
12:                 $\Sigma^S[i] = 1$, $\Sigma[i] = \Sigma[i] \oplus \alpha_0[MarkedBit[i]]$
13:         LinearTrans($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\Sigma$, $\Sigma^S$, $MarkedBit$)
14:         $flag$ =Sieve($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, TDTT)
15:         if $flag = 0$ then
16:             return 0                                          ▷ Discard the prefix pair
17:         else
18:             $a$ =the number of new deduced differences bits in $\alpha_0$ and $\beta_0$.
19:     return 1                                                  ▷ Accept the prefix pair
```

Deducing New Bit Differences from Bit Values. New bit differences can be deduced from bit values obtained in the value phase. For example, if the adversary finds that the input bits of an S-box are $x_1 = x'_1 = 0$ then from Table 3, $\delta_{in}[4] = \delta_{out}[4]$, where $\delta_{out}[4]$ can be derived from α_1. If $\delta_{in}[0]$ is known, $\delta_{in}[2]$ can be obtained by $\delta_{in}[2] = \delta_{in}[0] + \delta_{out}[0]$. The procedure of deducing new bit differences is done by checking the cases in Table 3 as shown in Algorithm 14 in Appendix A. The value phase of the deduce-and-sieve algorithm is shown in Algorithm 12.

The deduce-and-sieve algorithm is shown in Algorithm 13. With a prefix pair, the adversary first runs the difference phase. If there is a contradiction then the adversary discards the pair (Line 11); otherwise, the adversary starts the value phase (Line 7). If new bit differences are deduced in the value phase then the adversary runs the difference phase again; otherwise, she accepts the prefix pair (Line 9). As the size of the Keccak state is finite 1600 bits, the deduce-and-sieve algorithm will end in finite steps when no new bit differences can be obtained. In this way, the deduce-and-sieve algorithm has a practical complexity.

6.2 SAT

Some of the generated prefix pairs have been filtered by applying the deduce-and-sieve algorithm. The connectivity problems of the remaining prefix pairs are determined by using a SAT-solver called CryptoMiniSAT [26]. The recent version of CryptoMiniSAT accepts XOR clauses as input arguments to describe

Algorithm 11 Initialising the Value Phase

```
1: procedure InitialVP($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, $B$, $B'$, $B_S$, $B'_S$, FVDT)
2:     for each S-box do
3:         Deduce the output difference $\varDelta_{out}$ from $\alpha_1$.
4:         Deduce the truncated input difference $\varDelta_{in}^T$ from $\beta_0$ and $\beta_0^S$.
5:         $v$=FVDT($\varDelta_{in}^T$,$\varDelta_{out}$)
6:         if $T$=0 then
7:             continue
8:         else
9:             Find the indices of the five bits in the S-box as $i_0, i_1, \cdots, i_4$.
10:            for each integer $j \in [0, 5)$ do
11:                if $v_{j+5} = 1$ then                    ▷ $v_j$ is the $j$th bit of $v$.
12:                    $B[i_j] = v_j$, $B'[i_j] = v_j$, $B_S[i_j] = 1$, $B'_S[i_j] = 1$
```

Algorithm 12 Value Phase of the Derive-and-seive Algorithm

```
1: procedure VP($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, $A$, $A'$, $A_S$, $A'_S$, $B$, $B'$, $B_S$, $B'_S$, FVDT)
2:     InitialVP($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, $B$, $B'$, $B_S$, $B'_S$, FVDT)
3:     for each integer $i \in [0, 320)$ do
4:         CPkernel($A$, $B$, $A_S$, $B_S$, $i$)
5:         CPKernel($A'$, $B'$, $A'_S$, $B'_S$, $i$)
6:     $a$ =Update($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, $B$, $B'$, $B_S$, $B'_S$)          ▷ See Algorithm 14
7:     if $a = 0$ then
8:         return 0                         ▷ No new bit differences are deduced.
9:     else
10:        return 1                         ▷ New bit differences are deduced.
```

a SAT problem. It means that there is no need to convert XORs in the Keccak round function into the *conjunctive normal form* (CNF) as it was done in [17].

The procedure of converting the connectivity problem of a prefix pair (M_1, M'_1) into a SAT problem is shown in Algorithm 15 in Appendix D. It can be seen from Algorithm 15 that in order to convert the connectivity problem into a SAT problem the adversary just needs to assign chaining values as initial values and derive expressions of output differences of the first round. The non-linear term $v_0 = (v_1 + 1)v_2$ in χ can be converted into CNF clauses as $(\neg v_0 \vee \neg v_1 \vee v_2) \wedge (v_0 \vee v_1) \wedge (v_0 \vee \neg v_2)$, where v_0, v_1 and v_2 are internal variables (see Lines 28-30, 33-35).

7 Experiments and complexity analysis

We verify our work by implementing the attack and analysing its complexity. We generate $2^{41.3}$ prefix pairs fulfilling the conditions in Table 7 for one iteration. These prefix pairs are filtered with the deduce-and-sieve algorithm. According to our experiments, the filtering rate is $2^{-19.42}$. Thus, there are $2^{21.88}$ prefix pairs remained after applying our deduce-and-sieve algorithm. If we run the deduce-and-sieve algorithm without the value phase, the filtering rate becomes

Algorithm 13 Derive-and-seive Algorithm

```
 1: procedure DeriveSeive($M_1$, $M_1^{'}$, TDTT, FVDT)
 2:     $(A, A', A_S, A'_S, B, B', B_S, B'_S, \alpha_0, \alpha_0^S, \beta_0, \beta_0^S)$=Initial($M_1$, $M_1^{'}$)
 3:     $flag = 1$
 4:     while $flag$ do
 5:         $flag$ =DP($M_1$, $M_1^{'}$, $\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, TDTT)
 6:         if $flag$ then
 7:             $flag$ =VP($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, $A$, $A'$, $A_S$, $A'_S$, $B$, $B'$, $B_S$, $B'_S$, FVDT)
 8:             if $flag = 0$ then
 9:                 return 1                              ▷ Accept the prefix pair
10:         else
11:             return 0                                  ▷ Discard the prefix pair
```

$2^{-13.55}$. It can be seen that the value phase helps to improve the efficiency of the deduce-and-sieve algorithm by a factor of $2^{5.87}$.

The average running time of the deduce-and-sieve algorithm is 1.22×10^{-5}s for a prefix pair in average running on a single core of Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz. If we apply the SAT solver CryptoMiniSAT to determine the connectivity problem instead of using our deduce-and-sieve algorithm, the average running time of the SAT solver for every prefix pair is 0.31s on a single core of Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz. In conclusion, our approach outperforms the SAT solver by a factor of 2.54×10^4 on this special type of SAT problems.

As from the software performance figure [14], approximately 2^{21} times of Keccak 24-round permutations can be implemented in 1 second on a single core of Intel(R) Sandy Bridge(R) Core i5-2400 @ 3.10GHz. It indicates that one execution of 4-round SHA3-384 takes $4/24 \times 2^{-21} = 2^{-23.58}$s. Thus, our deduce-and-sieve algorithm on one prefix pair is approximately equivalent to $1.22 \times 10^{-5}/2^{-23.58} = 2^{7.25}$ SHA3-384 operations from our experiments. The average running time of the SAT solver for each remaining prefix pair after the deduce-and-sieve algorithm is 3.93s on the same platform, which is equivalent to $3.93/2^{-23.58} = 2^{25.55}$ SHA3-384 operations.

We use statistical methods to analyse the data complexity. We define a *semi-free n-bit chaining value collision attack* in which situation the adversary is assumed to have the capacity of modifying n-bit chaining values for each suffix message, where $n > 0$. From our experiments, it can be observed that there are 10.95 suffix seed pairs in average for each iteration to construct semi-free 14-bit chaining value collision attacks. Deriving the probability that there exists a solution for a connectivity problem in a non-statistical manner is an open problem.

Next, we estimate the probability that the suffix seed pair is still a seed pair for a semi-free $(n-1)$-bit chaining value collision attack, denoted as p_n, where $n \geq 1$. The probability is important in analysing the complexity of our attack. The probability p_n should be $\frac{1}{2}$, where $n > 0$. As discussed in Section 4.3, a suffix seed pair for constructing a semi-free n-bit chaining value collision attack should fulfill a system of linear equations combining Equation 3 and Equation 4. If the suffix seed pair is also a suffix pair for a semi-free $(n-1)$-bit chaining value collision attack, one extra constraint from that one extra bit chaining value should hold except for Equation 3 and Equation 4. Thus, the probability p_n is $\frac{1}{2}$, where $n > 0$.

To build a real collision attack, we need to collect 2^{14} suffix seed pairs for the semi-free 14-bit chaining value collision attack. Therefore, we need to generate $2^{41.3} \cdot 2^{14}/10.95 = 2^{51.85}$ prefix pairs. To generate these pairs, the adversary applies the hash table technique in Algorithm 3. As mentioned in Section 5.3, the time, data and memory complexity of the 1st block generation stage should be $2^{45.93}$.

In the 1-round SAT-based connector stage, the adversary applies the deduce-and-sieve algorithm to filter the $2^{51.85}$ prefix pairs. The time complexity is $2^{7.25} \cdot 2^{51.85} = 2^{59.1}$ and the memory complexity is negligible. Then, the adversary solves the connectivity problems for the remaining $2^{51.85} \cdot 2^{-19.42} = 2^{32.43}$ prefix pairs applying the SAT solver. The time complexity is $2^{32.43} \cdot 2^{25.55} = 2^{57.98}$. The memory cost of applying the SAT solver is also negligible from our experiments. Thus, the complexity of the second stage is $2^{59.64}$.

In the collision searching stage, the adversary solves the system of linear equations combining Equation 3 and Equation 4 with a prefix pair and a suffix seed pair gained from the previous stage, the time complexity of which is negligible. Then, the adversary searches the solutions of the linear equations for a suffix pair following the last 3-round differential characteristic of probability 2^{42}. Thus, the complexity of this stage is 2^{42}. Therefore, the time complexity is determined by the complexity of the second stage, which is $2^{59.64}$. The memory and data complexity are both $2^{45.93}$. Recall that the second stage includes two phases, which are applying the deduce-and-sieve algorithm to filter prefix pairs and solving the remaining connectivity problems with SAT solvers. It is also an open problem to find the optimal filtering rate of the deduce-and-sieve algorithm to balance the complexity of the two phases.

As the size of available memory we have is insufficient to generate $2^{51.77}$ prefix pairs in one run utilising a large hash table, we had to generate data in several iterations instead. Up till now, we have run 57 iterations on our platform. We denote the number of suffix seed pairs for constructing a semi-free n-bit chaining value collision attack in these iterations by Ω_n, where $6 \leq n \leq 14$. We also compute the fraction that a suffix seed pair for constructing a semi-free n-bit chaining value collision attack is still a seed pair for constructing a $(n-1)$-bit near collision, denoted by $\hat{p}_n$, where $7 \leq n \leq 14$. The experiment results are shown in Table 4. It can be seen that almost each $\hat{p}_n$ is close to 0.5, which verifies our claim that p_n should be 0.5. In these results, the best result is a suffix seed pair for constructing a semi-free 6-bit chaining value collision attack. We show our message pairs for a semi-free 6-bit chaining value collision in Table 8 in Appendix E.

n	14	13	12	11	10	9	8	7	6
Ω_n	624	291	133	66	29	13	12	7	3
$\hat{p}_n$	0.466	0.457	0.496	0.439	0.448	0.923	0.583	0.429	-

Table 4. Experiment Results

8 Conclusions

In this paper we describe a practical collision attack on 4-round SHA3-384. Our attack outperforms the previous collision attack, of complexity 2^{147}, proposed by Dinur et al. in [9]. Currently, our result includes a 6-bit near collision, but an adversary with slightly higher computing power can find a collision in practical time.

Although this work does not threaten the security of the full SHA-3 hash function, our results may be applied to analyse other sponge-based hash functions. The two cryptanalytic tools that we introduced in this work, namely, Truncated Difference Transition Table and Fixed Value Distribution Table, can be helpful in detecting non-random behaviour of S-boxes. This may be useful not only in analyses of other primitives but also for future designs of new secure non-linear layers in symmetric primitives.

With the deduce-and-sieve algorithm developed in this work, most of unsatisfiable cases in a class of SAT problems can be determined in a more efficient way than calling a SAT solver directly. The deduce-and-sieve algorithm may help to enhance the performance of a SAT solver for certain class of SAT problems.

References

1. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Koblitz, N. (ed.) Advances in Cryptology - CRYPTO '96. LNCS, vol. 1109, pp. 1–15. Springer (1996)
2. Biham, E., Chen, R., Joux, A.: Cryptanalysis of SHA-0 and Reduced SHA-1. J. Cryptol. **28**(1), 110–160 (2015)
3. Biham, E., Shamir, A.: Differential Cryptanalysis of the Data Encryption Standard. Springer (1993)
4. Chabaud, F., Joux, A.: Differential Collisions in SHA-0. In: Krawczyk, H. (ed.) Advances in Cryptology - CRYPTO '98. vol. 1462, pp. 56–71. Springer (1998)
5. Daemen, J.: Cipher and Hash Function Design Strategies Based on Linear and Differential Cryptanalysis, PhD thesis, Doctoral Dissertation, KU Leuven (1995)
6. Damgård, I.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89. LNCS, vol. 435, pp. 416–427. Springer (1989)
7. Diffie, W., Hellman, M.E.: New Directions in Cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976)
8. Dinur, I., Dunkelman, O., Shamir, A.: New Attacks on Keccak-224 and Keccak-256. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 442–461. Springer (2012)

9. Dinur, I., Dunkelman, O., Shamir, A.: Collision attacks on up to 5 rounds of SHA-3 using generalized internal differentials. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 219–240. Springer (2013)
10. Dinur, I., Morawiecki, P., Pieprzyk, J., Srebrny, M., Straus, M.: cube attacks and cube-attack-like cryptanalysis on the round-reduced keccak sponge function
11. Guido, B., Joan, D., Michaël, P., Assche, G.V.: Keccak Sponge Function Family Main Document `http://Keccak.noekeon.org/Keccak-main-2.1.pdf`
12. Guo, J., Liao, G., Liu, G., Liu, M., Qiao, K., Song, L.: Practical Collision Attacks against Round-Reduced SHA-3. J. Cryptol. **33**(1), 228–270 (2020)
13. Huang, S., Wang, X., Xu, G., Wang, M., Zhao, J.: Conditional Cube Attack on Reduced-Round Keccak Sponge Function. In: Coron, J., Nielsen, J.B. (eds.) Advances in Cryptology - EUROCRYPT 2017. LNCS, vol. 10211, pp. 259–288 (2017)
14. Keccak Team: Software performance figures. `https://keccak.team/sw_performance.html`, accessed: 2022-02-15
15. Knudsen, L.R.: Truncated and Higher Order Differentials. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008, pp. 196–211. Springer (1994)
16. Merkle, R.C.: A Certified Digital Signature. In: Brassard, G. (ed.) Advances in Cryptology - CRYPTO '89. LNCS, vol. 435, pp. 218–238. Springer (1989)
17. Morawiecki, P., Srebrny, M.: A SAT-based Preimage Analysis of Reduced Keccak Hash Functions. Inf. Process. Lett. **113**(10-11), 392–397 (2013)
18. Naya-Plasencia, M., Röck, A., Meier, W.: Practical Analysis of Reduced-Round Keccak. In: Bernstein, D.J., Chatterjee, S. (eds.) Progress in Cryptology - INDOCRYPT 2011. LNCS, vol. 7107, pp. 236–254. Springer (2011)
19. NIST: FIPS 202: SHA-3 Standard: Permutation-based Hash and Extendable-output Functions (2015)
20. NIST: FIPS 180-1: Secure Hash Standard (April 1995)
21. NIST: FIPS 180-2: Secure Hash Standard (August 2002)
22. NIST: FIPS: Secure Hash Standard (May 1993)
23. Qiao, K., Song, L., Liu, M., Guo, J.: New Collision Attacks on Round-Reduced Keccak. In: Coron, J., Nielsen, J.B. (eds.) Advances in Cryptology - EUROCRYPT 2017. Lecture Notes in Computer Science, vol. 10212, pp. 216–243 (2017)
24. Rivest, R.L.: The MD5 Message-Digest Algorithm. RFC **1321**, 1–21 (1992)
25. Song, L., Liao, G., Guo, J.: Non-full Sbox Linearization: Applications to Collision Attacks on Round-Reduced Keccak. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology - CRYPTO 2017. LNCS, vol. 10402, pp. 428–451. Springer (2017)
26. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer (2009)
27. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full SHA-1. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology - CRYPTO 2017. LNCS, vol. 10401, pp. 570–596. Springer (2017)
28. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the hash functions MD4 and RIPEMD. In: Cramer, R. (ed.) Advances in Cryptology - EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer (2005)
29. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R. (ed.) Advances in Cryptology - EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer (2005)
30. Wang, X., Yu, H., Yin, Y.L.: Efficient Collision Search Attacks on SHA-0. In: Shoup, V. (ed.) Advances in Cryptology - CRYPTO 2005. LNCS, vol. 3621, pp. 1–16. Springer (2005)

A Linear expressions of the output differences of χ with different conditions

Conditions	Linear Expressions	
$x_2 = x_2' = 0$	$\delta_{out}[1] = \delta_{in}[1] + \delta_{in}[3]$	$\delta_{out}[0] = \delta_{in}[0]$
$x_2 = x_2' = 1$	$\delta_{out}[1] = \delta_{in}[1]$	$\delta_{out}[0] = \delta_{in}[0] + \delta_{in}[1]$
$x_2 = 1,\ x_2' = 0$	$\delta_{out}[1] = \delta_{in}[1] + x_3'$	$\delta_{out}[0] = \delta_{in}[0] + x_1 + 1$
$x_2 = 0,\ x_2' = 1$	$\delta_{out}[1] = \delta_{in}[1] + x_3$	$\delta_{out}[0] = \delta_{in}[0] + x_1' + 1$
$x_3 = x_3' = 0$	$\delta_{out}[2] = \delta_{in}[2] + \delta_{in}[4]$	$\delta_{out}[1] = \delta_{in}[1]$
$x_3 = x_3' = 1$	$\delta_{out}[2] = \delta_{in}[2]$	$\delta_{out}[1] = \delta_{in}[1] + \delta_{in}[2]$
$x_3 = 1,\ x_3' = 0$	$\delta_{out}[2] = \delta_{in}[2] + x_4'$	$\delta_{out}[1] = \delta_{in}[1] + x_2 + 1$
$x_3 = 0,\ x_3' = 1$	$\delta_{out}[2] = \delta_{in}[2] + x_4$	$\delta_{out}[1] = \delta_{in}[1] + x_2' + 1$
$x_4 = x_4' = 0$	$\delta_{out}[3] = \delta_{in}[3] + \delta_{in}[0]$	$\delta_{out}[2] = \delta_{in}[2]$
$x_4 = x_4' = 1$	$\delta_{out}[3] = \delta_{in}[3]$	$\delta_{out}[2] = \delta_{in}[2] + \delta_{in}[3]$
$x_4 = 1,\ x_4' = 0$	$\delta_{out}[3] = \delta_{in}[3] + x_0'$	$\delta_{out}[2] = \delta_{in}[2] + x_3 + 1$
$x_4 = 0,\ x_4' = 1$	$\delta_{out}[3] = \delta_{in}[3] + x_0$	$\delta_{out}[2] = \delta_{in}[2] + x_3' + 1$
$x_0 = x_0' = 0$	$\delta_{out}[4] = \delta_{in}[4] + \delta_{in}[1]$	$\delta_{out}[3] = \delta_{in}[3]$
$x_0 = x_0' = 1$	$\delta_{out}[4] = \delta_{in}[4]$	$\delta_{out}[3] = \delta_{in}[3] + \delta_{in}[4]$
$x_0 = 1,\ x_0' = 0$	$\delta_{out}[4] = \delta_{in}[4] + x_1'$	$\delta_{out}[3] = \delta_{in}[3] + x_4 + 1$
$x_0 = 0,\ x_0' = 1$	$\delta_{out}[4] = \delta_{in}[4] + x_1$	$\delta_{out}[3] = \delta_{in}[3] + x_4' + 1$

Table 5. Linear Expressions of the Output Differences of χ with Known Input Bits in Different Positions

Algorithm 14 Deducing New Bit Differences

```
1: procedure UPDATE($\alpha_0$, $\beta_0$, $\alpha_1$, $\alpha_0^S$, $\beta_0^S$, $B$, $B'$, $B_S$, $B_S'$)
2:     for each integer $i \in [0, 1600)$ do
3:         if $\beta_0^S[i] = 0$ and $B_S[i] = 1$ and $B_S'[i] = 1$ then
4:             $\beta_0^S[i] = 1$, $\beta_0[i] = B[i] \oplus B'[i]$ .
5:     for each integer $i \in [0, 1600)$ do
6:         if $B_S[i] = 1$ and $B_S'[i] = 1$ then                    ▷ Property 5
7:             if $B[i] = 0$ and $B'[i] = 0$ then
8:                 Update $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)][\psi_2(i)]$, $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+2][\psi_2(i)]$ and $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+4][\psi_2(i)]$ according to the 1st row in Table 3
9:             else if $B[i] = 1$ and $B'[i] = 1$ then
10:                Update $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)][\psi_2(i)]$, $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+2][\psi_2(i)]$ and $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+4][\psi_2(i)]$ according to the 2nd row in Table 3
11:            else if $B[i] = 1$ and $B'[i] = 0$ then
12:                Update $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)][\psi_2(i)]$, $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+2][\psi_2(i)]$ and $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+4][\psi_2(i)]$ according to the 3rd row in Table 3
13:            else if $B[i] = 0$ and $B'[i] = 1$ then
14:                Update $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)][\psi_2(i)]$, $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+2][\psi_2(i)]$ and $\beta_0/\beta_0^S[\psi_0(i)][\psi_1(i)+4][\psi_2(i)]$ according to the 4th row in Table 3
15:    return $a$ =the number of new deduced differences bits in $\beta_0$.
```

B The 3-round differential characteristic

$\alpha_1(\Delta S_I)$	7c-bc4f5b4398--2	24-7de4bc9668--1	ac-2-95d32eb8---	d4-2e98975-68---	3c-57-6a-7f58---	1
	7c-bccf5b4398--2	24-fde4bc9e68--1	ac-2-95d32ef8---	c4-ae98975-68---	34147-6a-5f58---	
	7c-bc4f5b4398---	24-fda4bc9e68--1	ac-2-95d32eb8---	c4-ae9897d-68---	3c157-6a25f58---	
	7c-bc4f5bc398--2	24-fde4fc9668--1	ac-2-95d32eb8---	c4-ae98975-68---	3c157-6a-5f48---	
	7c-bc4f1b4398--2	24-fde4bc9e68--1	ac-2-95d3aeb8---	d4-ae98975868---	3c157-6a-5f58---	
β_1	----------------	----------------	----------------	----------------	----------------	2^{-26}
	---------1--8---	------------8-1-	--------------1-	------------8-1-	---------1------	
	--1------1------	---------1------	--1-------------	---------1------	--1-------------	
	--1---------8---	------------8---	--1-------------	----------------	--1---------8---	
	----------------	----------------	--------------1-	--------------1-	----------------	
β_2	----------------	8---------------	----------------	----------------	----------------	2^{-15}
	----------------	----------------	--------8-------	---------------1	----------------	
	----------------	----------------	----------------	---------------1	----------------	
	----------------	---------------1	----------------	----------------	---------------1	
	----------------	8---------------	--------8-------	----------------	----------------	
β_3	----------------	----------------	----------------	----------------	----------------	2^{-1}
	----------------	----------------	----------------	----2-----------	--------1-------	
	---------------1	------2---------	---------2------	----------------	----------------	
	----------------	----------------	-------------4--	----------------	----------------	
	----------------	--8-------------	----------------	----------------	---------------2	
$\alpha_4\ (\Delta\ S_O)$	----------------	----------------	----------------	----------------	----------------	–
	----------------	----------------	----------------	----2-----------	----?---1-------	
	---------------1	------2--------?	------?--2-----?	------?---------	----------------	
	----------------	----------------	-------------4--	-------------?--	-------------?--	
	---------------?	--8------------?	--?-------------	--?-------------	---------------2	

Table 6. The 3-round differential trail

A 3-round differential trail is shown in Table 6. The '?' in Table 6 means that the corresponding byte is unknown. In the last column, the number is the probability that the state transfers to the next state. For example, $\Pr(\alpha_1 \rightarrow \beta_1) = 1$, which is stated in the last entry of the first row. Therefore, the probability of the 3-round differential trail is 2^{-42}.

C Conditions on chaining values

Type-I Conditions	$\alpha_0[867] + \alpha_0[1187] = 0,\ \alpha_0[867] + \alpha_0[1187] = 0,\ \alpha_0[868] + \alpha_0[1188] = 0,$
	$\alpha_0[881] + \alpha_0[1201] = 0,\ \alpha_0[882] + \alpha_0[1202] = 0,\ \alpha_0[883] + \alpha_0[1203] = 0,$
	$\alpha_0[884] + \alpha_0[1204] = 0,\ \alpha_0[885] + \alpha_0[1205] = 0,\ \alpha_0[886] + \alpha_0[1206] = 0,$
	$\alpha_0[887] + \alpha_0[1207] = 0,\ \alpha_0[888] + \alpha_0[1208] = 0,\ \alpha_0[889] + \alpha_0[1209] = 0,$
	$\alpha_0[999] + \alpha_0[1319] = 0,\ \alpha_0[1000] + \alpha_0[1320] = 0,\ \alpha_0[1001] + \alpha_0[1321] = 0,$
	$\alpha_0[1036] + \alpha_0[1356] = 0,\ \alpha_0[1037] + \alpha_0[1357] = 0,\ \alpha_0[1038] + \alpha_0[1358] = 0,$
	$\alpha_0[1039] + \alpha_0[1359] = 0,\ \alpha_0[1040] + \alpha_0[1360] = 0,\ \alpha_0[1088] + \alpha_0[1408] = 0,$
	$\alpha_0[1148] + \alpha_0[1468] = 0,\ \alpha_0[1149] + \alpha_0[1469] = 0,\ \alpha_0[1150] + \alpha_0[1470] = 0,$
	$\alpha_0[1151] + \alpha_0[1471] = 0,\ \alpha_0[1216] + \alpha_0[1536] = 0,\ \alpha_0[1217] + \alpha_0[1537] = 0,$
	$\alpha_0[1218] + \alpha_0[1538] = 0,\ \alpha_0[1219] + \alpha_0[1539] = 0,\ \alpha_0[1220] + \alpha_0[1540] = 0,$
	$\alpha_0[1277] + \alpha_0[1597] = 0,\ \alpha_0[1278] + \alpha_0[1598] = 0,\ \alpha_0[1279] + \alpha_0[1599] = 0,$
	$\alpha_0[938] + \alpha_0[1578] = 0,\ \alpha_0[959] + \alpha_0[1279] = 1,\ \alpha_0[998] + \alpha_0[1318] = 1,$
	$\alpha_0[1147] + \alpha_0[1467] = 1,\ \alpha_0[836] + \alpha_0[1476] = 1,$
Type-II Conditions	$\alpha_0[952] + \alpha_0[1592] + \alpha_0[1373] + \alpha_0[1053] = 1$

Table 7. Set of parameters for the attack on 7-round Keccak-MAC-256

D Converting the Connectivity Problem into a SAT problem

Algorithm 15 Converting a connectivity problem into a SAT problem

```
Input: $M_1$, $M_1'$, $\alpha_1$
Output: An SAT problem $S_E$
1: $S_E = \emptyset$
2: $A = f(M_1||0)$, $A' = f(M_1'||0)$
3: $A[828] = A[828] \oplus 1$, $A'[828] = A'[828] \oplus 1$    ▷ Padding
4: $A[829] = A[829] \oplus 1$, $A'[829] = A'[829] \oplus 1$    ▷ Padding
5: $A[830] = A[830] \oplus 1$, $A'[830] = A'[830] \oplus 1$    ▷ Padding
6: for each integer $i \in [0, 828)$ do
7:     $A[i] = v(i)$, $A'[i] = v(i + 828)$
8: $c = 1656$
9: for each integer $i \in [0, 320)$ do    ▷ $\theta$
10:     $\Sigma[i] = v(c)$,
11:     Add an XOR clause $v(c)+A[i]+A[i+320]+A[i+640]+A[i+960]+A[i+1280] = 0$ to $S_E$
12:     $c = c + 1$
13:     $\Sigma'[i] = v(c)$,
14:     Add an XOR clause $v(c) + A'[i] + A'[i+320] + A'[i+640] + A'[i+960] + A'[i+1280] = 0$ to $S_E$
15:     $c = c + 1$
16: for each integer $i \in [0, 1600)$ do
17:     Add an XOR clause $v(c) + A[i] \oplus \Sigma[\phi_1(i)] \oplus \Sigma[\phi_2(i)] = 0$ to $S_E$
18:     $A[i] = v(c)$
19:     $c = c + 1$
20:     Add an XOR clause $v(c) + A'[i] \oplus \Sigma'[\phi_1(i)] \oplus \Sigma'[\phi_2(i)] = 0$ to $S_E$
21:     $A'[i] = v(c)$
22:     $c = c + 1$
23: $\rho(A)$, $\pi(A)$, $\rho(A')$, $\pi(A')$
24: for each integer $i \in [0, 5)$ do    ▷ $\chi$
25:     for each integer $j \in [0, 5)$ do
26:         for each integer $k \in [0, 64)$ do
27:             $B[i][j][k] = v(c)$    ▷ $B[i][j][k] = (A[i][j+1][k] + 1)A[i][j+2][k]$
28:             Add a clause $\neg v(c) \vee \neg A[i][j+1][k] \vee A[i][j+1][k]$ to $S_E$
29:             Add a clause $v(c) \vee A[i][j+1][k]$ to $S_E$
30:             Add a clause $v(c) \vee \neg A[i][j+2][k]$ to $S_E$
31:             $c = c + 1$
32:             $B'[i][j][k] = v(c)$    ▷ $B'[i][j][k] = (A'[i][j+1][k] + 1)A'[i][j+2][k]$
33:             Add a clause $\neg v(c) \vee \neg A'[i][j+1][k] \vee A'[i][j+1][k]$ to $S_E$
34:             Add a clause $v(c) \vee A'[i][j+1][k]$ to $S_E$
35:             Add a clause $v(c) \vee \neg A'[i][j+2][k]$ to $S_E$
36:             $c = c + 1$
37: for each integer $i \in [0, 1600)$ do
38:     Add an XOR clause $A[i] \oplus A'[i] \oplus B[i] \oplus B'[i] = \alpha_1[i]$ to $S_E$
```

E Semi-free Chaining Value Collision Messages

M_1	3f44b0f6d7099635 3ead03856312d62d 635f018370e46a8e	6a3d685cf646df2d bfe379262e56c467 ad3f8f3e0e7f3e03	900ecaf80bec3015 b42df34cbebeadd5 36dd5ac9c46ae7b3	ed4d0d18af18646a 85ec65a41546e49b	1a79525e62aa5f64 c3d2e6f410038d11
M_2	404a7ef90eb3d52b 805bcd6af83072a2 e03c2f16c447a314	5f5e0b31495be583 b13af0501e7f7916 fbae056f672fdcb9	7035405109c90eb3 d9e44137b6d155e8 ecb53c92855c8d9d	58ea891524eff68f 4489924da4ab43a5 1	62a0604a4804d991 f19059f8d30754bf
M_1'	bbe4518debe59253 93675cd9e7fe4605 f2d9d70f10332c27	da1895464a6afbb7 d453b4c818dfc8d2 9804aaba6c45be93	68427c475b7cb80e d953e4212c8282a7 eeabe6ed220fc6a6	bebc2cec2c568b43 cb22be4094203732	7a9a847a977e7ada b66dd121b1c586b
M_2'	6835bff9f37b323a 84c8b8f8553a5974 7dbe68bd8d92a300	6bfc9b8d0e5d3c1f c1dc6bd18408a113 f3de3f3287b3f3cd	e3c7f23bb07bb5b2 7cf27ae4c6484023 fe70dbd49cd7572	b67d8408aa9d8634 c2ada2969c37f057 3	cce98cbb4d471c56 eded0684eb43500f
H	bd06cbb861b6459f 4e9d212683ee8dd3	4cd5de69edf3c960	61e9efa56f5fe951	5bc4b75eb4293a4	bb0b9b830e682c81

Table 8. Semi-free 6-bit Chaining Value Collision Messages and Hash Value