Finding Collisions against 4-Round SHA-3-384 in Practical Time

Senyang Huang\(^1\,\text{†}\), Orna Agmon Ben-Yehuda\(^2\), Orr Dunkelman\(^3\) and Alexander Maximov\(^4\)

\(^1\) Department of Electrical and Information Technology, Lund University, Lund, Sweden
\(\text{senyang.huang@eit.lth.se}\)

\(^2\) Caesarea Rothschild Institute for Interdisciplinary Applications of Computer Science (CRI), University of Haifa, Haifa, Israel
\(\text{ladypine@gmail.com}\)

\(^3\) Department of Computer Science, University of Haifa, Haifa, Israel
\(\text{orrd@cs.haifa.ac.il}\)

\(^4\) Ericsson Research, Lund, Sweden
\(\text{alexander.maximov@ericsson.com}\)

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 of SHA-3 has attracted a lot of attention. Currently, the most powerful collision attack on SHA-3 is Jian Guo et al.’s linearisation technique. However, this technique is infeasible for variants with a smaller input space, such as SHA-3-384.

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

The resulting collision-finding algorithm on 4-round SHA-3-384 has a practical time complexity of \(2^{59.64}\) (and a memory complexity of \(2^{45.94}\)). This greatly improves upon the best known collision attack so far: Dinur et al. achieved an impractical \(2^{47}\) time complexity. Our attack does not threaten the security margin of the SHA-3 hash function. However, the tools developed in this paper could be used to analyse other cryptographic primitives as well as to develop new and faster SAT solvers.

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

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 Diffie and Hellman [DH76] suggesting signing a cryptographic hash value of a message rather than the message itself, hash functions became extremely useful in various cryptographic protocols: authentication (e.g., HMAC [BCK96]), password protection, commitment schemes, key

\[^{\text{†}}\text{This work was partially done when Senyang Huang was a post-doc researcher at the University of Haifa.}\]
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 [Riv92], 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 [NIS93]. Two years later, SHA was updated into the later named SHA-1 [NIS95] to prevent some attacks that were not disclosed to the public (but were later rediscovered by Chabaud and Joux [CJ98]). With the need for output sizes larger than SHA-1’s 160-bit, NIST published a new family of hash functions, called SHA-2, with output sizes of 224–512 bits [NIS02].

In 2005, Wang et al. [WLF+05, WY05, WYY05] broke several cryptographic functions. These fundamental works demonstrated the way to attack most of the existing hash functions using several techniques and ideas: using modular differences (i.e., using both an XOR difference and an additive difference); using multi-block collisions (i.e., collisions that span over several blocks, an idea independently discovered in [BCJ15]); and introducing 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, along with results on the Merkle-Damgård hash function [Dam89, Mer89], 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 [BDPA] was published as the new SHA-3 standard [NIS15].

Keccak [BDPA], 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 large 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.

The Keccak sponge function can be deployed in different modes, namely, keyed mode and unkeyed mode. Since the publication of Keccak in 2008, the analysis of both keyed mode and unkeyed mode Keccak has attracted considerable attention. For the keyed Keccak, a cube-like attack proposed by Dinur et al. [DMP+15] and a conditional cube attack proposed by Huang et al. [HWX+17] are the most powerful tools for analysing primitives based on the Keccak sponge function.

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 colliding pair should be computationally difficult for a secure hash function. In 2011, Naya-Plasencia et al. reported a collision attack on 2-round Keccak-512.¹, among several other practical attacks on the Keccak hash function [NRM11]. In 2012, Dinur et al. proposed practical collision attacks on 4-round Keccak-224/256 [DDS12]: they combined a 1-round connector with a 3-round low weight characteristic 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 [DDS13].

Currently, the most powerful tool for building a practical collision attack against the SHA-3 hash function is the linearisation technique [QSLG17, SLG17, GLL+20]. In [QSLG17], Qiao et al. followed the framework proposed by Dinur et al. in [DDS12] 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. Song et

¹SHA-3-n differs from Keccak-n only in the padding rule.
al. [SLG17] developed a new non-full linearisation technique to save degrees of freedom in the attack. Using this new technique, they launched several practical collision attacks on the Keccak family, such as 5-round SHA-3-224 and 5-round SHA-3-256. Jian Guo et al. recapped the two results [QSLG17, SLG17] in [GLL+20]. There was no further development after that. However, that technique cannot be directly applied to variants with a smaller input space, such as SHA-3-384 and SHA-3-512. It is because the appended conditions consume many degrees of freedom, which variants with a smaller input space cannot provide enough of.

Morawiecki et al. [MS13] applied a SAT solver to the analysis of an unkeyed mode modified Keccak, using different parameters than the recommended ones, to find the preimage of a hash value up to 3 rounds. However, the authors did not consider Keccak’s algebraic structure in their work, which reduces SAT solver’s power. In our collision attack, we combine algebraic non-random characteristics with a SAT solver to make full use of its efficiency.

Our Contributions
Our work extends previous results [DDS12] 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.

Inspired by the collision attacks proposed by Boissier et al. [BNR21] and the preimage attacks against Keccak-224/256 proposed in [LS19] by Li et al., we use more than a single block in the colliding message pair. Unlike the inner collision attacks against smaller Keccak variants in [BNR21], our technique can work on the outer part of a Keccak default variant. Namely, we noticed that often good input differences impose conditions on the input that 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 this restriction. This step increases the flexibility in choosing a differential characteristic. In addition, we use the first block to set some capacity bits to values that help the connectivity step.

In addition, we introduce another two techniques into collision attacks on Keccak. Our second contribution is to replace the linearisation connection phase that was used before in [GLL+20] with a SAT-connection phase. This idea is inspired by the dedicated collision attack against SHA-1 with aid of a SAT solver, proposed by Stevens et al. in [SBK+17]. Namely, we use SAT solvers to find message values that satisfy the required difference conditions while previous works [GLL+20, QSLG17, SLG17] did the connection from the input difference of the characteristic to the message conditions using linearisation. 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.

### Table 1: Summary of existing collision attacks on SHA-3.

| Rounds | Target | Complexity | Reference |
|--------|--------|------------|-----------|
| 4      | Keccak-224 | Practical | [DDS12]   |
| 5      | SHA-3-224 | Practical | [QSLG17]  |
| 4      | Keccak-256 | Practical | [DDS12]   |
| 5      | Keccak-256 | $2^{115}$ | [DDS13]   |
| 5      | SHA-3-256 | Practical | [QSLG17]  |
| 3      | Keccak-384 | Practical | [DDS13]   |
| 4      | Keccak-384 | $2^{147}$ | [DDS13]   |
| 4      | SHA-3-384 | $2^{59.64}$ | Section 7 |
| 2      | Keccak-512 | Practical | [NRM11]   |
| 3      | Keccak-512 | Practical | [DDS13]   |
The third contribution is the introduction of detection and sieving tools. They complete internal states more efficiently 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 orders of magnitude. We introduce a Truncated Difference Transform Table: for a given truncated differential transition, the table stores the 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 [BS93] stored in the difference distribution table also the values that correspond to the transition). Using these two tools enables, for each pair, the deduction of information needed to satisfy the differential characteristic.

We combine these ideas and produce the first practical attack that can find collisions in 4-round SHA-3-384. The expected running time of this attack is below $2^{60}$ (we remind the reader that the SHA-1 collision found by [SBK+17] used about $2^{63}$ computation). While we implemented the attack and verified it, our best result at the moment is a 4-bit semi-free internal collision, which is to date, the best known semi-free against SHA-3-384. We compare our result with previous results of collision attacks on SHA-3 in Table 1.

Moreover, our two-block collision attack can be extended to a multi-block attack, where the first few blocks can be chosen prefixes with meaningful information. This idea is inspired by the chosen-prefix collision attacks against MD5 [SLdW07] proposed by Stevens et al. and against SHA-1 [LP19] proposed by Leurent et al. A chosen-prefix collision attack is to find messages $(M, M')$ such that $H(P||M) = H(P'||M')$, where $P$ and $P'$ are chosen prefixes and $||$ denotes concatenation [SLdW07]. In this situation, the attacker’s task is then to find a collision while starting from a random difference in the internal state (due to the prefixes pair that is not controlled at all by the attacker). Chosen-prefix collision attacks are more difficult to mount but are stronger attacks more relevant to practice because the chosen prefixes can be arbitrary meaningful texts.

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 [BDPA, NIS15] for the complete Keccak specification.

The Keccak hash function works on a 1600-bit state $A$, which is treated as a three-dimensional array of bits, namely $A[5][5][64]$. As shown in Figure 1, the one-dimensional arrays $A[i][y][z]$, $A[x][y][z]$ and $A[x][y][i]$ are called a column, a row and a lane, respectively; the two-dimensional array $A[i][y][z]$ is called a slice. The coordinates are 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 $(64(5g + x) + z)$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 $A[\lfloor i/64 \rfloor][\lfloor i/64 \rfloor][i \mod 5]$, $A[i] = \lfloor i/32 \rfloor$, $\psi(0) = \lfloor i/32 \rfloor$, $\psi(1) = i \mod 5$, $\psi(2) = i \mod 64$, $\lfloor \cdot \rfloor$ is the floor function, and $0 \leq i < 1600$. We
also define a function $\phi_0$ 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 and the bitrate $r$ is 1152, 1088, 832, 576, respectively for Keccak-224, Keccak-256, Keccak-384 and Keccak-512.

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, $\theta, \rho, \pi, \chi$ and $\iota$. The round function is defined as $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$, with the following sub-functions:

$$
\theta : A[i][j][k] \leftarrow A[i][j][k] + \sum_{j'=0}^{4} A[i-1][j'][k] + \sum_{j'=0}^{4} A[i+1][j'][k-1],
$$

$$
\rho : A[i][j] \leftarrow A[i][j] \gg r[i, j], \quad r[i, j] \text{ is a constant},
$$

$$
\pi : A[j][2i + 3j] \leftarrow A[i][j],
$$

$$
\chi : A[i][j] \leftarrow A[i][j] + (A[i+1][j] + 1)A[i+2][j][k],
$$

$$
\iota : A[0][0] \leftarrow A[0][0] + RC_{i, r}, \quad RC_{i, r} \text{ is the } i, r \text{ round constant},
$$

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

The operation $\theta$ diffuses the state. In the operation $\theta$, 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 $\rho$ and $\pi$ 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 $\pi \circ \rho$. We denote by $L$ the first three linear operations $\theta, \rho$ and $\pi$, which we call a half round. We rewrite the expression of $L$ in Equation 1:

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

In Equation 1, $A$ is the input state of $L$ while $B$ is the output state. $\text{col}[\phi_1(\sigma^{-1}(i))]$ and $\text{col}[\phi_2(\sigma^{-1}(i))]$ 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

![Figure 1: Terminologies in Keccak.](image-url)
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 \( \overline{M} = M/10 \times 1 \).

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 \( \overline{M} = M/0110 \times 1 \).

### 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 column parity kernel (CP-kernel) equation: for states in which all columns have even parity, \( \theta \) is the identity [BDPA]. This property has been widely used in cryptanalysis of Keccak. E.g., the attacks in [HWX+17] use it to control the diffusion of cube variables.

**Property 1.** (CP-kernel Equation) For every \( i \)-th and \( j \)-th 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 \( \chi \), we first recall the definition of the difference distribution table (DDT) of \( \chi \) [BS93]. The operation \( \chi \) is applied to each row of the state independently, and can be regarded as an \( S \)-box. In differential cryptanalysis proposed by Biham and Shamir in [BS93], 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.\(^2\) We present an important property of the solution set as follows.

**Property 2.** ([Dae95, DDS12, BDPA]) 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.

**Property 3.** If \( \delta_{out} = 0x1 \) then \( \delta_{in}[0] = 1 \).

**Property 4.** If \( \delta_{out} = 0x3 \) then \( \delta_{in}[1] \oplus \delta_{in}[3] = 1 \).

We summarise all cases with special output differences in Table 2 when the output differences in Property 3 and Property 4 are shifted to the right.

**Table 2:** Summary of conditions for special output differences of \( \chi \).

| 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             | \( \delta_{in}[0] \oplus \delta_{in}[2] = 1 \) |

\(^2\)The idea of the solution set first appeared in Biham and Shamir’s original work on differential cryptanalysis.
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 $y_4'y_3'y_2'y_1'y_0$ and $y_4'y_3'y_2'y_1'y_0$.

Property 5. ([GLL+20]) 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 directly follows from the algebraic relation between the input and the output of $\chi$. 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 |
|------------|--------------------|
| $x_1 = x'_1 = 0$ | $\delta_{out}[0] = \delta_{in}[0] + \delta_{in}[2]$ | $\delta_{out}[4] = \delta_{in}[4]$ |
| $x_1 = x'_1 = 1$ | $\delta_{out}[0] = \delta_{in}[0]$ | $\delta_{out}[4] = \delta_{in}[4] + \delta_{in}[0]$ |
| $x_1 = 1$, $x'_1 = 0$ | $\delta_{out}[0] = \delta_{in}[0] + x_2$ | $\delta_{out}[4] = \delta_{in}[4] + x_0 + 1$ |
| $x_1 = 0$, $x'_1 = 1$ | $\delta_{out}[0] = \delta_{in}[0] + x_2$ | $\delta_{out}[4] = \delta_{in}[4] + x'_0 + 1$ |

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. They were constructed by Guo et al. utilising an algebraic and differential hybrid method [GLL+20], which follows the 1-round connector technique proposed by Dinur et al. in [DDS12].

The framework of the attack against SHA-3-n is shown in Figure 2. Given an $n_2$-round high-probability differential characteristic $\Delta S_T \to \Delta S_O$ with the first $n$ bits of the output difference $\Delta S_O$ 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 $((M_1, M'_1)|R_{n_1}, (M_1)|0) \oplus R_{n_1}, (M'_1)|0 = \Delta S_I$, where $\Delta S_I$ is the input difference of the differential characteristic. In the second stage, the adversary finds a colliding pair following the $n_2$-round differential characteristic by searching through pairs of messages obtained in the first stage.

![Figure 2: Overview of Guo et al.’s $(n_1 + n_2)$-round collision attacks.](image_url)

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 SHA-3-384 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 A new framework for a collision attack against 4-round SHA-3-384

In this section we introduce a new framework for a collision attack that overcomes drawbacks in the techniques of both Dinur et al. and Guo et al. 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.

![Figure 3: Framework of our attack.](image-url)

The first blocks $M_1$ and $M'_1$ in Figure 3 are called prefixes. A four-round Keccak permutation, denoted as $f$, is operated on $M_1$ and $M'_1$, before absorbing the next blocks $M_2$ and $M'_2$, respectively. The second blocks $M_2$ and $M'_2$ are called suffixes. 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. Note that a block of SHA-3-384 is with size of $1600 - 384 \times 2 = 832$ bits. 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 $(c + 4)$ 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 Table 4. The ‘?’ in Table 4 means that the corresponding nibble is unknown. 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 $\alpha_i \xrightarrow{L} \beta_i \xrightarrow{\chi} \alpha_i$, where $i \geq 1$. Let $\alpha_0$ be the input difference of the second block after adding message blocks as shown in Figure 3. Next, we overview the three stages.
We develop a new 1-round SAT-based connector that replaces Guo et al.’s linearisation technique, and removes the constraint 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 \( \alpha_1 \). This is a connectivity problem, 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

\[
R(f(M_1||0) \oplus (M_2||0)) \oplus R(f(M'_1||0) \oplus (M'_2||0)) = \alpha_1.
\]

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 \( \alpha_0 \) to \( \alpha_1 \), where the last 1600 - 828 = 772 bits of \( \alpha_0 \) 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 problem for the remaining pairs. For a prefix pair \((M_1, M'_1)\), if there exists a suffix pair \((M_2, M'_2)\) 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 \((M_2, M'_2)\), 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 \(M'_2\). We will discuss later that searching the affine subspace of \(M_2\) for the colliding pair is equivalent to searching the affine subspace of \(M'_2\).

Given a pair of prefix and suffix seeds \((M_2, M'_2)\), the input difference of the operation \(\chi\), denoted as \(\beta_0\), can be deduced by computing \(L(f(M_1)||0) \oplus L(f(M'_1)||0)\). Let \(x\) be a vector of bit values before \(\chi\), i.e. \(x = L(f(M_1)||0) \oplus (M'_2||0)\). By Property 2, given \(\beta_0\) and \(\alpha_1\), all the linear equations on the input affine subspaces of active S-boxes in the first round can be derived and expressed as

\[
A_1 \cdot x = b_1,
\]

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 constraints that \(x\) needs to fulfill are that given \(M_1\), the chaining values are prefixed:

\[
A_2 \cdot x = b_2,
\]

where \(A_2\) is a submatrix of \(L^{-1}\) and \(b_2\) is the vector of those prefixed chaining values computed from \(f(M_1)||0\). 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 the colliding pair that follows the 3-round differential characteristic depicted in Figure 3.

Searching the affine subspace of \(M_2\), denoted as \(W_1\), for the colliding pair is equivalent to searching the affine subspace of \(M'_2\), denoted as \(W_2\). As discussed above, \(x' = L(f(M'_1)||0) \oplus (M'_2||0)\) satisfies a system of linear equations combined with two parts. The first part is in a similar form to Equation 3, which is

\[
A_1 \cdot x' = b_1.
\]

The second part is written as:

\[
A_2 \cdot x' = b'_2,
\]

where \(A_2\) is a submatrix of \(L^{-1}\) and \(b_2\) is the vector of those prefixed chaining values computed from \(f(M'_1)||0\).

For each \(x \in L(W_1)\), \(x \oplus \beta_0\) is in \(L(W_2)\). We will show next that \(x \oplus \beta_0\) fulfills both Equation 5 and Equation 6. As \(x \in W_1\), \((x, x \oplus \beta_0)\) should follow the differential characteristic \(\beta_0 \rightarrow \alpha_1\), which indicates that \(x \oplus \beta_0\) fulfills Equation 5. As \(L\) is linear, the following equation should work:

\[
L^{-1}(x \oplus \beta_0) = L^{-1}(x) \oplus L^{-1}(\beta_0) = L^{-1}(x) \oplus \alpha_0.
\]
Algorithm 1 Deriving Linear Conditions

Input: $\alpha_1$
Output: Set of Linear Conditions $S_A$

1: Compute the output difference for each S-box from $\alpha_1$.
2: Initialise the system of equations $E$.
3: $S_0 = \emptyset$, $S_1 = \emptyset$, $S_A = \emptyset$.
4: for each S-box do
5:   $\delta_{\text{out}}$ is the output difference of the S-Box.
6:   if $\delta_{\text{out}} = 0$ then
7:      for each bit in the S-box do
8:         Add the corresponding equation $\beta_0[i_1] = 0$ to both $E$ and $S_0$.
9:   else if $\delta_{\text{out}} \in \{0x1, 0x2, 0x4, 0x8, 0x10\}$ then
10:      Check Table 2 and add the corresponding equation $\beta_0[i_1] = 1$ to $E$ and $S_1$.
11:   else if $\delta_{\text{out}} \in \{0x3, 0x6, 0xc, 0x11, 0x18\}$ then
12:      Check Table 2 and add the corresponding equation $\beta_0[i_1] + \beta_0[i_2] = 1$ to $E$.
13: Substitute the variables $\beta_0[0, \cdots, 1599]$ in $E$ by $\alpha_0[0, \cdots, 1599]$ according to the expression of $\theta$.
14: Reduce $E$ to its row echelon form.
15: for each equation in $E$ do
16:   if all the variables in the equation are from $\{\alpha_0[828], \cdots, \alpha_0[1599]\}$ then
17:      Add this equation to $S_A$
18: return $S_A$

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 the chaining values

For the connectivity problem to have at least one solution (at least one pair of compatible suffixes), chaining values must follow certain linear conditions. We demonstrate how to comply with these conditions in chaining values. With $\alpha_1$, the adversary obtains the output difference of each S-box from $\alpha_1$. There are three types of output differences from which the adversary can derive conditions on $\beta_0$. The other output differences do not derive conditions on $\beta_0$. These cases are listed as follows:

- **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 $0x18$.

The adversary can derive conditions from the three types of output differences by applying linear algebra. The procedure is shown in Algorithm 1. From Line 6 to Line 12, the adversary obtains the three types of output differences from $\alpha_1$ and writes the corresponding conditions on the input differences according to Property 3 and Property 4. Then, the adversary transforms the system of equations $E$ in the terms of $\alpha_0$ according
to $\theta$ operation and reduces $E$ to its row echelon form. At last, the adversary checks each equation in $E$ and obtains linear conditions on chaining values.

### 5.2 How to construct a 3-round differential characteristic

The 3-round differential characteristic in our attack adapts the second characteristic in [GLL+20, 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 bits have a zero difference. 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 $\beta_1$ is fixed, the 3-round differential characteristic is determined. We choose $\beta_1$ 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 negligible.

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

The difference $\alpha_2$ 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 no less than $(2^{-4})^8 \cdot 2^{-16} = 2^{-48}$. The dimension of the affine subspace in the collision searching stage should be larger than 48. The probability of the first round transition should not be larger than $2^{-828+48} = 2^{-780}$. As the average probability of a nonzero differential transition in DDT 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 the first message blocks is infeasible.

To extend the 2-round characteristic, the adversary picks a $\beta_1$ at random, which is compatible with $\alpha_2$ from the second characteristic in [GLL+20, Table 9]. Then, the adversary computes $\alpha_1$ as $L^{-1}(\beta_1)$. Given $\alpha_1$, the adversary obtains the number of active S-boxes in the first round. With Algorithm 1, the adversary deduces conditions on the chaining values. If the two criteria are satisfied, the adversary outputs a 3-round characteristic; otherwise, the adversary picks another compatible $\beta_1$ and continues the procedure.

In our differential characteristic presented in Table 4, there are 228 active S-boxes in the first round. Applying Algorithm 1 on the differential characteristic, there are 39 conditions on the chaining values. These conditions are listed in Appendix B. The probability of the differential characteristic is $2^{-42}$.

We note that the differential characteristic searching process needs to be run only once and its complexity is of polynomial time from our experiment. Furthermore, the found differential characteristic may not be the most optimal one. Finding the optimal differential characteristic is thus an open problem.
Algorithm 2 Generating Prefix Pairs

1: procedure GPP($n$)  
2:   Constant XOR $\Sigma$=0x7c00000000  
3:   $S_P = \emptyset$  
4:   Initialise an array Counter of length $2^{39}$ with zeros.  
5:   for each integer $i \in [0, 2^n)$ do  
6:     Randomly pick a message $M$ of 832 bits and compute the value string $c$.  
7:     HashTable[$c$][Counter[$c$]]=$M$  
8:     Increase Counter[$c$] by 1.  
9:   for each integer $i \in [0, 2^n)$ do  
10:      if $i < i \oplus \Sigma$ then  
11:         for each integer $j \in [0, \text{Counter}[i])$ do  
12:             for each integer $k \in [0, \text{Counter}[i \oplus \Sigma])$ do  
13:                 $S_P = S_P \cup \{(\text{HashTable}[i][j], \text{HashTable}[i \oplus \Sigma][k])\}$  
14:      return $S_P$

5.3 Generating prefix pairs fulfilling the requirements

We explain the generation procedure of prefix pairs fulfilling the 39 conditions in Table 9. 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 additional definitions.

We define a constant XOR as a binary string $c_38c_37\cdots c_1c_0$, where $c_i$ is the sum in the $i$-th condition, $0 \leq i \leq 38$. In our case, the constant XOR is 0x7c00000000 from the conditions in Table 9. The conditions are the sums of differences in certain bit positions of the input state of the second block. To check whether a given prefix pair $(M_1, M_1')$ satisfies the conditions, the adversary first computes sums of binary values in these positions, which we call by the value string and records with a 39-bit string. Then, the adversary sums up value strings and checks whether the result equals the constant XOR 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 2. First, the adversary generates $2^n$ first-block messages $M$ of length 832 bits and computes corresponding value strings $c$. The adversary places $M$ into the $c$th row of the hash table. Thereafter, the adversary searches through the hash table for prefix pairs that satisfy the constraints (Lines 8 to 12).

Algorithm 2 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.

It should be noted that the prefixes generated in Algorithm 2 can be extended to messages of length $832n_b$ bits, where $n_b$ is the number of blocks and $n_b \geq 1$. The procedure of generating multi-block prefix pairs is shown in Algorithm 16 of Appendix C. As shown in Algorithm 16, the adversary starts from arbitrary chosen messages $(P,P')$ as part of the prefixes in the first $(n_b-1)$ blocks. The remaining block is picked randomly to fulfill the linear conditions on the chaining values. Thus, our attack can be extended to a chosen-prefix collision attack like the works of [SLdW07, LP19].

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 2. Then, for each remaining prefix
Algorithm 3 Initial Phase of the Deduce-and-sieve Algorithm

1: procedure INITIAL($M_1$, $M'_1$)
2: \hspace{1cm} $A = f(M_1||0)$, $A' = f(M'_1||0)$
3: \hspace{1cm} for each integer $i \in [0,1600)$ do
4: \hspace{2cm} if $i \geq 828$ then
5: \hspace{3cm} $\alpha_0[i] = A[i] \oplus A'[i]$, $\alpha_0^S[i] = 1$, $A_S[i] = 1$, $A'_S[i] = 1$
6: \hspace{2cm} else
7: \hspace{3cm} $\alpha_0[i] = 0$, $\alpha_0^S[i] = 0$, $A[i] = 0$, $A'[i] = 0$, $A_S[i] = 0$, $A'_S[i] = 0$ \hspace{0.5cm} $\triangleright$ The second block is controlled by the adversary.
8: \hspace{1cm} if $i \in S_0$ then
9: \hspace{2cm} $\beta_0[i] = 0$, $\beta_0^S[i] = 1$
10: \hspace{2cm} else if $i \in S_1$ then
11: \hspace{3cm} $\beta_0[i] = 1$, $\beta_0^S[i] = 1$
12: \hspace{2cm} else
13: \hspace{3cm} $\beta_0^S[i] = 0$
14: \hspace{1cm} for each integer $i \in [0,320)$ do
15: \hspace{2cm} $\Sigma[i] = 0$, $\Sigma^S[i] = 0$
16: return $A$, $A'$, $A_S$, $A'_S$, $\alpha_0$, $\alpha_0^S$, $\beta_0$, $\beta_0^S$

pair, the connectivity problem is solved 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 $\beta_0$ 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 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 properties of Keccak’s 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 3), 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 $i$-th 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 $\beta_0$ as a constant vector. Two indicator vectors $\alpha_0^S$ and $\beta_0^S$ record whether the bit difference in a corresponding position is known.

In the deduce-and-sieve algorithm, we use the vector $\Sigma$ of length 320 to record sums of five bit differences in each column. In the beginning, each entry of the indicator vector $\Sigma^S$ 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 \(\alpha_0\) 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 \(\beta_0\) 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 \(\alpha_0\) and \(\beta_0\). We investigate the differential transition of the Keccak S-box and develop a tool called Truncated Difference Transform Table (TDTT), which is inspired by truncated differential cryptanalysis, proposed by Knudsen in [Knu94]. 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 transition from the input difference of the first round \(\alpha_0\) to the output difference \(\alpha_1\). Before we define the TDTT of an S-box, we first introduce truncated difference in Definition 2. \(^3\)

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

We use a \(2n\)-bit integer \(a||b\), where \(a, b \in \mathbb{F}_2^n\), to represent a truncated difference. Let \((a_{n-1}, \ldots, a_0)\) and \((b_{n-1}, \ldots, b_0)\) 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 \(0 \leq i < n\), \(a_i \leq b_i\). 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 \(\{d \in \mathbb{F}_2^n \mid d_i = b_{n+i} \text{ if } a_i = 1, 0 \leq i < n\}\). 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||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 001001||00001 in the binary representation. A complete transforming behaviour of the differences of an S-box can be described by its TDTT:

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

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

where \(\Delta_{in}^T\) is the new truncated input difference, \(\Delta_{in}^T, \Delta_{out} \in \mathbb{F}_2^n\) and \(\Delta_{out} \in \mathbb{F}_2^2\).

In case of Property 3, it can be observed that TDTT\((0||0, 1)\) is 0x1||0x1.

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

\(^3\)We alert the reader that the original definition in [Knu94] allowed for sets of differences. We use the common interpretation that characterizes the set by bits set to 0.
Algorithm 4 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 $a \in [0, 2^n)$ 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 $D = \{\Delta_m \in \mathbb{F}_2^n | \text{DDT}(\Delta_m, b) \neq 0 \} \cap \{\Delta_m \in \mathbb{F}_2^n | \Delta_m \leq a\}$
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:             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. The difference $\beta[\sigma(i_4)]$ 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 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 5.

Figure 4: Deducing new differences from the CP-kernel equations.

Figure 5: Representatives when the sum state is 1.
**Algorithm 5** Deducing New Differences in the $i$-th Column

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

![Figure 6: Representatives when the sum state is 2.](image)

**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:

$$\beta_0[i] = \alpha_0[\sigma^{-1}(i)] \oplus \Sigma[\phi_1(\sigma^{-1}(i))] \oplus \Sigma[\phi_2(\sigma^{-1}(i))],$$  \hspace{2em} (8)

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 8 is unknown, its value can be deduced. Before we show the technique of deducing new difference applying Equation 8, 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 sum state that encodes the three states: 1 for the first case, 2 for the second case, and 0 for the third case.

Representatives, when the sum state is 1, are shown in Figure 5.\textsuperscript{4} 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 of the cases when the sum state is 2 are shown in Figure 6.\textsuperscript{5} In these three 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.

\textsuperscript{4}We only show a representative in Figure 5 (b). The other cases can be obtained by permuting the bits in the two columns simultaneously.

\textsuperscript{5}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 6 Computing the $i$-th Column Sum

1: **procedure** ColumnSum($\alpha_0$, $\beta_0$, $\alpha^S_0$, $\beta^S_0$, $i$, $\Sigma$, $\Sigma^S$, 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: Set $\Sigma^S[i] = 1$

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: Set $\Sigma^S[i] = 2$

9: Record the index of the corresponding marked bit in MarkedBit[$i$]

10: else

11: Set $\Sigma^S[i] = 0$

The procedure for computing the $i$-th column sum is shown in Algorithm 6. The adversary finds the states of the 10 related bits from $\alpha_0$ and $\beta_0$. If the states of 10 related bits match one of the representatives in Figure 5, the column sum is known. The adversary computes the sum and updates the sum state $\Sigma[i]$ with 1. If the states of 10 related bits match 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 the column sums, the adversary can obtain new bit differences from the expression of $\theta$. To be more specific, if there is only one variable is unknown in Equation 8, the value of it can be easily deduced. The procedure is shown in Algorithm 7. There are 5 situations in which new differences can be deduced. In the two situations shown in Figure 6, marked bits are obtained and corresponding column sums are updated.

**Sieving prefix pairs with TDTT.** The sieving procedure applying the TDTT of the Keccak S-box is shown in Algorithm 8. With $\beta_0$ and $\beta^S_0$, the adversary 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 $\beta_0$ from the TDTT. Once a bit difference is obtained, the adversary checks the related CP-kernel equations to deduce new bit differences in $\alpha_0$ and $\beta_0$, as shown in Lines 13-15 in Algorithm 8.

Let us summarise the difference phase of the deduce-and-sieve algorithm in Algorithm 9. 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 9. 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 uses another algebraic property of the Keccak round function to deduce new input bit values of $\chi$ 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.
Algorithm 7 Deducing New Differences from the Expression of L

1: procedure LINEARTRANS(α₀, β₀, α₀S, β₀S, Σ, ΣS, MarkedBit)
2:     for each integer i ∈ [0, 1600) do
3:         i₀ = σ⁻¹(i), i₁ = φ₁(i₀), i₂ = φ₂(i₀)
4:         Deduce the expression of β₀[i] = α₀[i₀] ⊕ Σ[i₁] ⊕ Σ[i₂]
5:         if β₀S[i] = 0 ∧ α₀S[i₀] = 1 ∧ ΣS[i₁] = 1 ∧ ΣS[i₂] = 1 then
6:             Set β₀[i] = α₀[i₀] ⊕ Σ[i₁] ⊕ Σ[i₂], β₀S[i] = 1
7:             Call CPKERNEL(α₀, β₀, α₀S, β₀S, φ₀(i₀))
8:         else if β₀S[i] = 1 ∧ α₀S[i₀] = 0 ∧ ΣS[i₁] = 1 ∧ ΣS[i₂] = 1 then
9:             Set α₀[i₀] = β₀[i] ⊕ Σ[i₁] ⊕ Σ[i₂], α₀S[i₀] = 1
10:            Call CPKERNEL(α₀, β₀, α₀S, β₀S, φ₀(i₀))
11:         else if β₀S[i] = 1 ∧ α₀S[i₀] = 1 ∧ ΣS[i₁] = 0 ∧ ΣS[i₂] = 1 then
12:             Set Σ[i₁] = β₀[i] ⊕ α₀[i₀] ⊕ Σ[i₂], ΣS[i₁] = 1
13:         else if β₀S[i] = 1 ∧ α₀S[i₀] = 1 ∧ ΣS[i₁] = 2 ∧ ΣS[i₂] = 1 then
14:             Set α₀[MarkedBit[i₁]] = β₀[i] ⊕ α₀[i₀] ⊕ Σ[i₁] ⊕ Σ[i₂]
15:             Set α₀S[MarkedBit[i₁]] = 1
16:             Set Σ[i₁] = α₀[MarkedBit[i₁]] ⊕ Σ[i₁], ΣS[i₁] = 1
17:             Call CPKERNEL(α₀, β₀, α₀S, β₀S, φ₀(MarkedBit[i₁]))
18:         else if β₀S[i] = 1 ∧ α₀S[i₀] = 1 ∧ ΣS[i₁] = 1 ∧ ΣS[i₂] = 0 then
19:             Set Σ[i₂] = β₀[i] ⊕ α₀[i₀] ⊕ Σ[i₁], ΣS[i₂] = 1
20:         else if β₀S[i] = 1 ∧ α₀S[i₀] = 1 ∧ ΣS[i₁] = 1 ∧ ΣS[i₂] = 2 then
21:             Set α₀[MarkedBit[i₂]] = β₀[i] ⊕ α₀[i₀] ⊕ Σ[i₁] ⊕ Σ[i₂]
22:             Set α₀S[MarkedBit[i₂]] = 1
23:             Set Σ[i₂] = α₀[MarkedBit[i₂]] ⊕ Σ[i₂], ΣS[i₂] = 1
24:             Call CPKERNEL(α₀, β₀, α₀S, β₀S, φ₀(MarkedBit[i₂]))

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-kernel 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 the details of the FVDT of an n-bit S-box, we first define a solution set of a truncated input difference Δ₈ and an output difference Δ₉ as follows:

Definition 4. The solution set of a truncated input difference Δ₈ and an output difference Δ₉ is

\[ S_T(\Delta_8^T, \Delta_9) = \bigcup_{\Delta_8 \leq \Delta_8^T} \{ (a, a \oplus \Delta_8) | S(a) \oplus S(a \oplus \Delta_8) = \Delta_9 \}, \]

where Δ₈ ∈ \( \mathbb{F}_2^n \), Δ₈ ∈ \( \mathbb{F}_2^n \) and Δ₉ ∈ \( \mathbb{F}_2^n \).

The solution set of the truncated input difference Δ₈ and the output difference Δ₉ is a generalisation of the solution set of the input difference Δ₈ and the output difference Δ₉ of regular DDTs. From Definition 4, it is a union of solution sets of the input difference Δ₈ and the output difference Δ₉, where Δ₈ \( \preceq \Delta_8^T \).

We observe that for the truncated input difference Δ₈ and the output difference Δ₉, some bits of the pairs in the solution set \( S_T(\Delta_8^T, \Delta_9) \) may be constants. For example,
Algorithm 8 Discarding Prefix Pairs with TDTT

1: procedure STEVE($\alpha_0$, $\beta_0$, $\alpha_1$, $\beta_0^S$, $\beta_0^T$, 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:     $T \leftarrow$ 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:            Set $\beta_0[i_j] = T_j$, $\beta_0^S[i_j] = 1$ \\
15:                $\triangleright$ $T_j$ is the $j$-th bit of $T$.
16:            Call CPKernel($\alpha_0$, $\beta_0$, $\alpha_0^S$, $\beta_0^S$, $\phi_0(\sigma^{-1}(i_j)))$
17:        end if
18:       end for
19:    end if
20: end for

when the truncated input difference $\Delta_{in}^T$ is 0xc2, the covered differences are 0x2, 0x3, 0xa, 0xb, 0x12, 0x13, 0x1a, and 0x1b. If the output difference $\Delta_{out}$ is 0x1, the compatible differences are 0xb and 0x1b. The solution set is $S_T(0xc2,0x1) = \{0x0, 0x3, 0x8, 0xb\} \cup \{0x1, 0x1a\}$. 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 with $\Delta_{in}^T$ and $\Delta_{out}$, we use a 2n-bit integer $a\|b$, called as fixed point, to record constant values and their corresponding positions, where $a, b \in F_2^n$. 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 $\Delta_{in}^T$ and an output difference $\Delta_{out}$, 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 $\Delta_{in}^T \in F_2^{2n}$, $\Delta_{out} \in F_2^n$ and $v$ is the fixed point with respect to $\Delta_{in}^T$ and $\Delta_{out}$.

The adversary uses the FVDT of the Keccak S-box to initialise the value phase. The process is shown in Algorithm 10.

**Deducing new values from CP-kernel equations.** Similar 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 $M_i$ and $M'_i$, 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.

**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
Algorithm 9 Difference Phase of the Deduce-and-sieve 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:    Call 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:        Call 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:        Set $\Sigma^S[i] = 1$, $\Sigma[i] = \Sigma[i] \oplus \alpha_0[MarkedBit[i]]$
13:      Call 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  \triangleright Discard the prefix pair
17:    else
18:      $a =$ the number of new deduced differences bits in $\alpha_0$ and $\beta_0$.
19:  return 1  \triangleright Accept the prefix pair

bits of an S-box are $x_1 = x'_1 = 0$ then from Table 3, $\delta_m[4] = \delta_{out}[4]$, where $\delta_{out}[4]$ can be derived from $\alpha_1$. If $\delta_m[0]$ is known, $\delta_m[2]$ can be obtained by $\delta_m[2] = \delta_m[0] + \delta_{out}[0]$. The procedure of deducing new bit differences is done by checking the cases in Table 3 as shown in Algorithm 15 in Appendix A. The value phase of the deduce-and-sieve algorithm is shown in Algorithm 11.

The deduce-and-sieve algorithm is shown in Algorithm 12. 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 terminate after a finite number of steps when no new bit differences can be obtained. In this way, the deduce-and-sieve algorithm has a practical complexity, which is discussed further in Section 7.

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 [SNC09]. The recent version of CryptoMiniSAT accepts XOR clauses as input arguments to describe a SAT problem. Hence, there is no need to convert XORs in the Keccak round function into the conjunctive normal form (CNF) as it was done in [MS13].

The procedure of converting the connectivity problem of a prefix pair ($M_1, M'_1$) into a SAT problem is shown in Algorithm 17 in Appendix D. It can be seen from Algorithm 17 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. For example, the non-linear term $v_0 = (v_1 + 1)v_2$ in $\chi$ can be converted into CNF clauses as $(\neg v_0 \lor \neg v_1 \lor v_2) \land (v_0 \lor v_1) \land (v_0 \lor \neg v_2)$, where $v_0$, $v_1$, and $v_2$ are internal variables (see Lines 28-30, 33-35).
Algorithm 10 Initialise the Value Phase

1: procedure InitialVP($\alpha_0, \beta_0, \alpha_1, \alpha_0^S, \beta_0^S, B, B', B_S, B'_S, \text{FVDT}$)
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: $v=\text{FVDT}(\Delta_{in}^t, \Delta_{out})$
6: if $v=0$ 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 \Comment{$v_j$ is the $j$-th bit of $v$.}
12: Set $B[i_j] = v_j$, $B'[i_j] = v_j$, $B_S[i_j] = 1$, $B'_S[i_j] = 1$
13: else
14: Set $B[i_j] = 0$, $B'[i_j] = 0$, $B_S[i_j] = 0$, $B'_S[i_j] = 0$

Algorithm 11 Value Phase of the Deduce-and-sieve Algorithm

1: procedure VP($\alpha_0, \beta_0, \alpha_1, \alpha_0^S, \beta_0^S, A, A', A_S, A'_S, B, B', B_S, B'_S, \text{FVDT}$)
2: Call InitialVP($\alpha_0, \beta_0, \alpha_1, \alpha_0^S, \beta_0^S, B, B', B_S, B'_S, \text{FVDT}$)
3: for each integer $i \in [0, 320)$ do
4: Call CPKernel($A, B, A_S, B_S, i$)
5: Call CPKernel($A', B', A'_S, B'_S, i$)
6: $a=\text{Update}(\alpha_0, \beta_0, \alpha_1, \alpha_0^S, \beta_0^S, B, B', B_S, B'_S) \Comment{See Algorithm 15}$
7: if $a = 0$ then
8: return 0 \Comment{No new bit differences are deduced.}
9: else
10: return 1 \Comment{New bit differences are deduced.}

7 Experiments and complexity analysis

We verified our work by implementing the attack and analysing its complexity. We generated $2^{41.3}$ prefix pairs fulfilling the conditions in Table 9 for one iteration. Most of these prefix pairs were filtered with the deduce-and-sieve algorithm. According to our experiments, the filtering rate is $2^{-19.42}$. Thus, about $2^{21.88}$ 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 is only $2^{-13.55}$. It can be seen that the value phase helps to improve the efficiency of the deduce-and-sieve algorithm by a factor of $2^{5.87}$.

It is also interesting to note that when the capacity increases, the filtering rate of the deduce-and-sieve algorithm decreases sharply. For example, if the capacity is increased by 16 bits to 788 bits, the filtering rate of the deduce-and-sieve algorithm is $2^{-26.1}$. Thus, it is difficult to investigate the property of the remaining pairs using a statistical manner when the input space is even smaller. To build a collision attack against SHA-3-512, an improved deduce-and-sieve algorithm is needed.

The average running time of the deduce-and-sieve algorithm is $1.22 \times 10^{-5}$s for a prefix pair 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 the same platform. In conclusion, our approach outperforms the SAT solver by a factor of $2.54 \times 10^4$ on this special type of SAT problems.
Algorithm 12 Deduce-and-sieve Algorithm

1: procedure DERIVE-SIEVE($M_1, M'_1, TDTT, FVDT$)
2: ($A, A', A_S, A'_S, B, B', B_S, B'_S, \alpha_0, \alpha'_0, \beta_0, \beta'_0$) ← INITIAL($M_1, M'_1$)
3: flag = 1
4: while flag do
5: flag = DP($M_1, M'_1, \alpha_0, \beta_0, \alpha'_1, \alpha'_0, \beta'_0$, TDTT)
6: if flag then
7: flag = VP($\alpha_0, \beta_0, \alpha'_1, \alpha'_0, \beta'_0, A, A', A_S, A'_S, B, B', B_S, B'_S, FVDT$)
8: if flag = 0 then
9: return 1
10: else
11: return 0

As from the software performance figure\textsuperscript{6}, approximately $2^{21}$ calls for 24-round Keccak 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 SHA-3-384 takes $4/24 \times 2^{21} = 2^{-23.58}$s. Thus, our deduce-and-sieve algorithm on one prefix pair is approximately equivalent to $10^{-5}/2^{-23.58} = 2^{7.25}$ SHA-3-384 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 $3.93/2^{-23.58} = 2^{25.55}$ SHA-3-384 operations.

We use statistical methods to analyse the data complexity. Deriving the probability that there exists a solution for a connectivity problem in a non-statistical manner is an open problem. We define a semi-free n-bit internal collision attack in which situation the adversary is assumed to have the capacity of modifying n-bit chaining values for each suffix message, where $n > 0$. The corresponding connectivity problem is called semi-free n-bit internal connectivity problem. From our experiments, there are 11.07 suffix seed pairs on average for each iteration to construct semi-free 14-bit internal collision attacks.

To analyse the complexity of our attack, we show an important property of the connectivity problems in Observation 1.

Observation 1. The probability that a semi-free internal n-bit connectivity problem is still satisfiable as a semi-free $(n-1)$-bit internal connectivity problem, denoted as $p_n$, is approximately $\frac{1}{2}$, where $n \geq 1$.

Observation 1 is discovered through experiments. We estimate $p_n$ by computing the ratio of suffix seed pairs, which are still compatible for constructing a semi-free $(n-1)$-bit internal collision attack, in the seed pairs for semi-free $(n-1)$-bit internal collision attacks. The ratio is denoted as $\hat{p}_n$. The procedure of estimating $p_n$ for each $n \in [1, 14]$ is shown in Algorithm 13.

As for each $n \in [1, 14]$, we randomly generate 25600 internal state pairs $(m, m')$ to the second block such that $R(m) \oplus R(m') = \alpha_1$. To be more specific, we randomly pick $z \in F^{1600}_2$ and obtain an internal state pair to the second block, which is $(m, m') = (R^{-1}(z), R^{-1}(z \oplus \alpha_1))$. If we assume that the last 1600 - $(828 + n)$ = 772 - $n$ bits of $(m, m')$ are the chaining values, $(m, m')$ is a suffix seed pair for constructing a semi-free n-bit internal collision attack. From line 6 to 8 in Algorithm 13, we modify the $(827 + n)$-th bits of $m$ and $m'$ and denote the new pair as $(m_1, m'_1)$. We call the SAT solver to check whether $(m_1, m'_1)$ is a suffix seed pair for a semi-free $(n-1)$-bit internal collision attack. Finally, Algorithm 13 returns the ratio $\hat{p}_n$. The experimental results are shown in Table 5. It can be seen that each $\hat{p}_n$ is close to 0.5, where $n \in [1, 14]$.

It follows from Observation 1 that to build a real collision attack, we need to collect $2^{14}$ suffix seed pairs for the semi-free 14-bit internal collision attack. Therefore, we need

\textsuperscript{6}Software performance figures, \url{https://keccak.team/sv_performance.html}. 

The time complexity is $2^{62}$ Finding Collisions against 4-Round SHA-3-384 in Practical Time

Estimated $\hat{p}_n$, where $n \in [1,14]$.

| $n$ | 14 | 13 | 12 | 11 | 10 | 9  | 8  |
|-----|----|----|----|----|----|----|----|
| $\hat{p}_n$ | 0.591 | 0.628 | 0.534 | 0.635 | 0.533 | 0.540 | 0.491 |

| $n$ | 7  | 6   | 5   | 4   | 3   | 2   | 1   |
|-----|----|-----|-----|-----|-----|-----|-----|
| $\hat{p}_n$ | 0.578 | 0.534 | 0.524 | 0.500 | 0.497 | 0.516 | 0.543 |

Algorithm 13 Estimating $p_n$

Input: $n$

Output: Estimated $\hat{p}_n$

1: $ctr = 0$
2: for each integer $i \in [0, 25600)$ do
3: Randomly pick $z \in \mathbb{F}_2^{1600}$
4: Compute an internal input pair to the second block $(m,m') = (R^{-1}(z), R^{-1}(z \oplus \alpha_1))$
5: for each integer $j \in [0, 4)$ do
6: $m_1 = m, m'_1 = m'$
7: Add the $(827 + n)$-th bit of $m_1$ with the first bit of $j$
8: Add the $(827 + n)$-th bit of $m'_1$ with the second bit of $j$
9: Call CryptoMiniSAT to determine whether $(m_1, m'_1)$ is a suffix seed pair for a semi-free $(n-1)$-bit internal collision attack
10: if $(m_1, m'_1)$ is a seed pair then
11: $ctr = ctr + 1$
12: return $\hat{p}_n = ctr/(4 \times 256000)$

Table 5:

To generate $2^{41.3} \cdot 2^{14}/11.07 = 2^{51.83}$ prefix pairs. To generate these pairs, the adversary applies the hash table technique in Algorithm 2. As mentioned in Section 5.3, the time, data and memory complexity of the 1st block generation stage should be $2^{45.92}$. The whole procedure of the attack is summarised in Algorithm 14.

In the 1-round SAT-based connector stage, the adversary applies the deduce-and-sieve algorithm to filter the $2^{51.83}$ prefix pairs. The time complexity is $2^{25.25} \cdot 2^{51.83} = 2^{59.1}$ and the memory complexity is negligible. Then, the adversary solves the connectivity problems for the remaining $2^{51.83} \cdot 2^{-19.42} = 2^{32.41}$ prefix pairs applying the SAT solver. The time complexity is $2^{32.41} \cdot 2^{21.55} = 2^{57.96}$. The memory cost of applying the SAT solver is also negligible from our experiments. Thus, the complexity of the second stage is $2^{59.1} + 2^{57.96} \approx 2^{59.64}$.

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 $2^{-42}$. From our experiments, the solution space of the linear equations is typically with a dimension of 100, which is sufficiently large to find a colliding pair. Thus, the complexity of this stage is $2^{42}$.

The time complexity of our collision attack is determined by the complexity of the second stage, which is $2^{59.64}$. The memory and data complexity are both $2^{45.92}$. 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 by choosing a proper $\alpha_0$.

As the size of available memory we have is insufficient to generate $2^{51.83}$ prefix pairs in one run utilising a large hash table, we have to generate data in several iterations instead. Each iteration takes 43 hours on our platform, which is equal to $43 \times 256 = 1.1 \times 10^4$
Algorithm 14 The procedure of the attack

**Output:** \((M_1||M_2, M'_1||M'_2)\) such that the hash value of the two strings is the same.

1. \(S_P = GPP(45.92)\) ▷ 1st block generation stage
2. for each pair \((M_1, M'_1)\) in \(S_P\) do ▷ 1-round SAT-based connector stage
3. if DERIVESIEVE\((M_1, M'_1, \text{TDTT, FVDT})\) then
4. \(S_E = \text{SAT}(M_1, M'_1, \text{e})\) ▷ See Algorithm 17
5. Solve the SAT problem \(S_E\) with the SAT solver CryptoMiniSAT
6. if \(S_E\) is satisfiable then
7. Output the suffix seed \((M_2, M'_2)\) from CryptoMiniSAT
8. Go to Step No.11
9. else continue
10.
11. Solve the system of linear equations \(E\) combining Equation 3 and Equation 4 with \((M_1, M'_1)\) and \((M_2, M'_2)\). ▷ Collision searching stage
12. Search the solutions of \(E\) for a colliding pair \((M_2, M'_2)\).
13. return \((M_1||M_2, M'_1||M'_2)\)

After a detailed analysis, we find that 11.07 suffix seed pairs in each iteration for semi-free 14-bit internal collision attacks. To find a collision pair of 4-round SHA-3-384, approximately \(2^{14}/11.07 = 1480\) iterations are needed, which is 1.63 \(\times 10^7\) core-hours. To be more specific, the total running time is around 7.3 years on our platform.

Up till now, we have run 106 iterations on our platform. In these iterations, the best result is a suffix seed pair for constructing a semi-free 4-bit internal collision attack, which is consistent with our estimation. We show our message pairs for a semi-free 4-bit internal collision in Table 6. We also show the suffix seed pairs in Table 7.

Table 6: Semi-free 4-bit Internal Collision Messages and Hash Value.

| \(M_1\) | \(M_2\) | \(S_{M_1}\) | \(S_{M_2}\) | \(H\) |
|-----|-----|-----|-----|-----|
| a0b837ea6d3a833a | 3a67013dd90c8c1a | a0afd65757f0e1dd | dc42016f089ee317 | 3f5732121a0fbfccdd |
| 5b3f3de5af8b3513 | 2539995219a2ce0b | 91218525188f2fc1 | 5d5192031990a2ef | 6be5f0a54d323649 |
| 43b80b30c11643c8 | 6e17f6154a55be62 | 8d943ff358e8dd8a | ed3e58fde7229fec | 455abf407629c87d |
| 455130b30c11643c | 6de32803877902bb | 9eb057116c73aaca | 102d4731eb8f9549 | 8170fc1f64fbf10d |
| 0c3cb176013dd90c | 314cd39448837878 | a7570403a2a21341 | f4544f840a2f65a7 | 10418d4373762 |

Table 7: Suffix Seed Pair for Semi-free 4-bit Internal Collision Attack.

| \(M_2\) | \(M'_2\) | \(H\) |
|-----|-----|-----|
| 2621484657a54 | 02b2981f35e261a0f | b68531f7b5a02ev6 |
| 26d6557b1579123 | a2f548d4569038 | 5af8e84373762 |

8 Conclusions

In this paper we describe a practical collision attack on 4-round SHA-3-384. Our attack outperforms the previous collision attack, of complexity \(2^{47}\), proposed by Dinur et al. in [DDS13]. Currently, our result includes a semi-free 4-bit internal 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 crypto-
analytic tools that we introduced in this work, namely, Truncated Difference Transform Table and Fixed Value Distribution Table, can be helpful in detecting non-random behaviour of S-boxes. These tools may be useful in analyses of other primitives with the sponge construction, for example, Keccak with smaller states [BDPA], Xoodyak [DHP +20], Gimili [BKL +17] and etc. The tools may be also useful 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.

Acknowledgments

Some of the work of the first author was done while visiting Jian Guo’s group at Nanyang Technological University. We also thank Jian Guo, Yi Tu, Itai Dinur, Thomas Johansson, Jeff Yan and Matúš Nemec for their thoughtful insights.

The research is supported in part by the Center for Cyber, Law, and Policy in conjunction with the Israel National Cyber Bureau in the Prime Minister’s Office and by the Israeli Science Foundation through grants No. 880/18 and 3380/19. The first author is funded by an ELLIT project. Part of the computations are enabled by resources provided by LUNARC. We also thank Qian Guo for granting us the access of the resources through his project.

References

[BCJ15] Eli Biham, Rafi Chen, and Antoine Joux. Cryptanalysis of SHA-0 and Reduced SHA-1. J. Cryptol., 28(1):110–160, 2015.

[BCK96] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, Advances in Cryptology - CRYPTO ’96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings, volume 1109 of Lecture Notes in Computer Science, pages 1–15. Springer, 1996.

[BDPA] Guido Bertoni, Joan Daemen, Michæl Peeters, and Gilles Van Assche. Keccak Sponge Function Family Main Document. http://Keccak.noekeon.org/Keccak-main-2.1.pdf.

[BKL 17] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashiw Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. Gimli: A Cross-Platform Permutation. In Wieland Fischer and Naofumi Homma, editors, Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings, volume 10529 of Lecture Notes in Computer Science, pages 299–320. Springer, 2017.

[BNR21] Rachelle Heim Boissier, Camille Noûs, and Yann Rotella. Algebraic Collision Attacks on Keccak. IACR Trans. Symmetric Cryptol., 2021(1):239–268, 2021.

[BS93] Eli Biham and Adi Shamir. Differential Cryptanalysis of the Data Encryption Standard. Springer, 1993.

[CJ98] Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In Hugo Krawczyk, editor, Advances in Cryptology - CRYPTO ’98, 18th Annual
Senyang Huang, Orna Agmon Ben-Yehuda, Orr Dunkelman, Alexander Maximov

International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998. Proceedings, volume 1462 of Lecture Notes in Computer Science, pages 56–71. Springer, 1998.

[Dae95] Joan Daemen. Cipher and Hash Function Design Strategies Based on Linear and Differential Cryptanalysis, PhD thesis, Doctoral Dissertation, KU Leuven. 1995.

[Dam89] Ivan Damgård. A Design Principle for Hash Functions. In Gilles Brassard, editor, Advances in Cryptology - CRYPTO ’89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20–24, 1989, Proceedings, volume 435 of Lecture Notes in Computer Science, pages 416–427. Springer, 1989.

[DDS12] Itai Dinur, Orr Dunkelman, and Adi Shamir. New Attacks on Keccak-224 and Keccak-256. In Anne Canteaut, editor, Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers, volume 7549 of Lecture Notes in Computer Science, pages 442–461. Springer, 2012.

[DDS13] Itai Dinur, Orr Dunkelman, and Adi Shamir. Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials. In Shiho Moriai, editor, Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers, volume 8424 of Lecture Notes in Computer Science, pages 219–240. Springer, 2013.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. IEEE Trans. Inf. Theory, 22(6):644–654, 1976.

[DHP+20] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodyak, a Lightweight Cryptographic Scheme. IACR Trans. Symmetric Cryptol., 2020(S1):60–87, 2020.

[DMP+15] Itai Dinur, Pawel Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michal Straus. Cube Attacks and Cube-Attack-Like Cryptanalysis on the Round-Reduced Keccak Sponge Function. In Elisabeth Oswald and Marc Fischlin, editors, Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I, volume 9056 of Lecture Notes in Computer Science, pages 733–761. Springer, 2015.

[GLL+20] Jian Guo, Guohong Liao, Guozhen Liu, Meicheng Liu, Kexin Qiao, and Ling Song. Practical Collision Attacks against Round-Reduced SHA-3. J. Cryptol., 33(1):228–270, 2020.

[HWX+17] Senyang Huang, Xiaoyun Wang, Guangwu Xu, Meiqin Wang, and Jingyuan Zhao. Conditional Cube Attack on Reduced-Round Keccak Sponge Function. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II, volume 10211 of Lecture Notes in Computer Science, pages 259–288, 2017.

[Knu94] Lars R. Knudsen. Truncated and Higher Order Differentials. In Bart Preneel, editor, Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings, volume 1008 of Lecture Notes in Computer Science, pages 196–211. Springer, 1994.
[LP19] Gaëtan Leurent and Thomas Peyrin. From Collisions to Chosen-Prefix Collisions Application to Full SHA-1. In Yuval Ishai and Vincent Rijmen, editors, Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III, volume 11478 of Lecture Notes in Computer Science, pages 527–555. Springer, 2019.

[LS19] Ting Li and Yao Sun. Preimage Attacks on Round-Reduced Keccak-224/256 via an Allocating Approach. In Yuval Ishai and Vincent Rijmen, editors, Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III, volume 11478 of Lecture Notes in Computer Science, pages 556–584. Springer, 2019.

[Mer89] Ralph C. Merkle. A Certified Digital Signature. In Gilles Brassard, editor, Advances in Cryptology - CRYPTO ’89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings, volume 435 of Lecture Notes in Computer Science, pages 218–238. Springer, 1989.

[MS13] Pawel Morawiecki and Marian Srebrny. A SAT-based Preimage Analysis of Reduced Keccak Hash Functions. Inf. Process. Lett., 113(10-11):392–397, 2013.

[NIS93] NIST. FIPS: Secure Hash Standard. May 1993.

[NIS95] NIST. FIPS 180-1: Secure Hash Standard. April 1995.

[NIS02] NIST. FIPS 180-2: Secure Hash Standard. August 2002.

[NIS15] NIST. FIPS 202: SHA-3 Standard: Permutation-based Hash and Extendable-output Functions. August 2015.

[NRM11] María Naya-Plasencia, Andrea Röck, and Willi Meier. Practical Analysis of Reduced-Round Keccak. In Daniel J. Bernstein and Sanjit Chatterjee, editors, Progress in Cryptology - INDOCRYPT 2011 - 12th International Conference on Cryptology in India, Chennai, India, December 11 -14, 2011. Proceedings, volume 7107 of Lecture Notes in Computer Science, pages 236–254. Springer, 2011.

[QSLG17] Kexin Qiao, Ling Song, Meicheng Liu, and Jian Guo. New Collision Attacks on Round-Reduced Keccak. In Jean-Sébastien Coron and Jesper B. Nielsen, editors, Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III, volume 10212 of Lecture Notes in Computer Science, pages 216–243, 2017.

[Riv92] Ronald L. Rivest. The MD5 Message-Digest Algorithm. RFC, 1321:1–21, 1992.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The First Collision for Full SHA-1. In Jonathan Katz and Hovav Shacham, editors, Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I, volume 10401 of Lecture Notes in Computer Science, pages 570–596. Springer, 2017.
[SLdW07] Marc Stevens, Arjen K. Lenstra, and Benne de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In Moni Naor, editor, Advances in Cryptology - EUROCRYPT 2007, 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007, Proceedings, volume 4515 of Lecture Notes in Computer Science, pages 1–22. Springer, 2007.

[SLG17] Ling Song, Guohong Liao, and Jian Guo. Non-full Sbox Linearization: Applications to Collision Attacks on Round-Reduced Keccak. In Jonathan Katz and Hovav Shacham, editors, Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II, volume 10402 of Lecture Notes in Computer Science, pages 428–451. Springer, 2017.

[SNC09] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In Oliver Kullmann, editor, Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings, volume 5584 of Lecture Notes in Computer Science, pages 244–257. Springer, 2009.

[WLF+05] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In Ronald Cramer, editor, Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings, volume 3494 of Lecture Notes in Computer Science, pages 1–18. Springer, 2005.

[WY05] Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In Ronald Cramer, editor, Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings, volume 3494 of Lecture Notes in Computer Science, pages 19–35. Springer, 2005.

[WYY05] Xiaoyun Wang, Hongbo Yu, and Yiqun Lisa Yin. Efficient Collision Search Attacks on SHA-0. In Victor Shoup, editor, Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings, volume 3621 of Lecture Notes in Computer Science, pages 1–16. Springer, 2005.
A Linear expressions of the output differences of χ with different conditions

Table 8: Linear expressions of the output differences of χ with known input bits in different positions.

| Conditions | Linear Expressions |
|------------|--------------------|
| x_2 = x_1' = 0 | δ_{in[1]} = δ_{in[1]} + δ_{in[3]} | δ_{out[0]} = δ_{in[0]} |
| x_2 = x_1' = 1 | δ_{in[1]} = δ_{in[1]} | δ_{out[0]} = δ_{in[0]} + δ_{in[1]} |
| x_2 = 1, x_1' = 0 | δ_{in[1]} = δ_{in[1]} + x_5 | δ_{out[0]} = δ_{in[0]} + x_5 + 1 |
| x_2 = 0, x_1' = 1 | δ_{in[1]} = δ_{in[1]} + x_5 | δ_{out[0]} = δ_{in[0]} + x_5' + 1 |
| x_2 = x_1' = 0 | δ_{in[2]} = δ_{in[2]} + δ_{in[4]} | δ_{out[1]} = δ_{in[1]} |
| x_2 = x_1' = 1 | δ_{in[2]} = δ_{in[2]} | δ_{out[1]} = δ_{in[1]} + δ_{in[2]} |
| x_2 = 1, x_1' = 0 | δ_{in[2]} = δ_{in[2]} + x_4 | δ_{out[1]} = δ_{in[1]} + x_4 + 1 |
| x_2 = 0, x_1' = 1 | δ_{in[2]} = δ_{in[2]} + x_4 | δ_{out[1]} = δ_{in[1]} + x_4' + 1 |
| x_2 = x_1' = 0 | δ_{in[3]} = δ_{in[3]} + δ_{in[4]} | δ_{out[2]} = δ_{in[2]} |
| x_2 = x_1' = 1 | δ_{in[3]} = δ_{in[3]} | δ_{out[2]} = δ_{in[2]} + δ_{in[3]} |
| x_2 = 1, x_1' = 0 | δ_{in[3]} = δ_{in[3]} + x_5 | δ_{out[2]} = δ_{in[2]} + x_5 + 1 |
| x_2 = 0, x_1' = 1 | δ_{in[3]} = δ_{in[3]} + x_5 | δ_{out[2]} = δ_{in[2]} + x_5' + 1 |
| x_2 = x_1' = 0 | δ_{in[4]} = δ_{in[4]} + δ_{in[4]} | δ_{out[3]} = δ_{in[3]} |
| x_2 = x_1' = 1 | δ_{in[4]} = δ_{in[4]} | δ_{out[3]} = δ_{in[3]} + δ_{in[4]} |
| x_2 = 1, x_1' = 0 | δ_{in[4]} = δ_{in[4]} + x_4 | δ_{out[3]} = δ_{in[3]} + x_4 + 1 |
| x_2 = 0, x_1' = 1 | δ_{in[4]} = δ_{in[4]} + x_4 | δ_{out[3]} = δ_{in[3]} + x_4' + 1 |

Algorithm 15 Deducing New Bit Differences

1: procedure UPDATE(α_0, β_0, α_1, α_0' \bar{B}', B, B', B_B')
2: for each integer i \in [0, 1600) do
3: if β_0'[i] = 0 and B_B[i] = 1 and B_B'[i] = 1 then
4: Set β_0'[i] = 1, β_0'[i] = B'[i] \oplus B'[i] .
5: for each integer i \in [0, 1600) do
6: if B'[i] = 1 and B_B'[i] = 1 then
7: if B[i] = 0 and B'[i] = 0 then
8: Update β_0/β_0'[\bar{v}_0(i)][\psi_1(i)][\psi_2(i)], β_0/β_0'[\bar{v}_0(i)][\psi_1(i)] + 2[\psi_2(i)] and β_0/β_0'[\bar{v}_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 β_0/β_0'[\bar{v}_0(i)][\psi_1(i)][\psi_2(i)], β_0/β_0'[\bar{v}_0(i)][\psi_1(i)] + 2[\psi_2(i)] and β_0/β_0'[\bar{v}_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_B'[i] = 0 then
12: Update β_0/β_0'[\bar{v}_0(i)][\psi_1(i)][\psi_2(i)], β_0/β_0'[\bar{v}_0(i)][\psi_1(i)] + 2[\psi_2(i)] and β_0/β_0'[\bar{v}_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_B'[i] = 1 then
14: Update β_0/β_0'[\bar{v}_0(i)][\psi_1(i)][\psi_2(i)], β_0/β_0'[\bar{v}_0(i)][\psi_1(i)] + 2[\psi_2(i)] and β_0/β_0'[\bar{v}_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 β_0.
B Conditions on chaining values

Table 9: Conditions on chain values.

| Expression                  | Value       |
|-----------------------------|-------------|
| $a_0[939] + a_0[1579] = 0$  | $a_0[967]$  |
| $a_0[881] + a_0[1201] = 0$ | $a_0[1187]$ |
| $a_0[884] + a_0[1204] = 0$ | $a_0[1205]$ |
| $a_0[887] + a_0[1207] = 0$ | $a_0[1208]$ |
| $a_0[999] + a_0[1319] = 0$ | $a_0[1000]$ |
| $a_0[1036] + a_0[1356] = 0$| $a_0[1037]$ |
| $a_0[1039] + a_0[1359] = 0$| $a_0[1040]$ |
| $a_0[1148] + a_0[1468] = 0$| $a_0[1150]$ |
| $a_0[1151] + a_0[1471] = 0$| $a_0[1216]$ |
| $a_0[1218] + a_0[1538] = 0$| $a_0[1219]$ |
| $a_0[1277] + a_0[1597] = 0$| $a_0[1278]$ |
| $a_0[968] + a_0[1578] = 0$ | $a_0[959]$  |
| $a_0[1147] + a_0[1467] = 1$| $a_0[836]$  |
| $a_0[1592] + a_0[1593] + a_0[1053] = 1$ | |

C Generating Multi-Block Prefix Pairs

Algorithm 16 Generating $n_b$-Block Prefix Pairs

**Input:** Chosen $(n_b - 1)$-Block Prefixes $(P, P')$

**Output:** Set of $n_b$-Block Prefixes $S_P$

1. Constant XOR $\Sigma$=0x7e00000000
2. $S_P = \emptyset$
3. Initialise two arrays, $Cntr_A$ and $Cntr_B$ of length $2^{39}$ with zeros.
4. for each integer $i \in [0, 2^n)$ do
5. Randomly pick a 1-block message $M$ of 832 bits and compute the value string $c$ on $(P||M)$.
6. Set $HT_A[c][Cntr_A[c]] = M$ and increase $Cntr_A[c]$ by 1.
7. Compute the value string $c'$ on $(P'||M)$.
8. Set $HT_B[c'][Cntr_B[c']] = M$ and increase $Cntr_B[c']$ by 1.
9. for each integer $i \in [0, 2^n)$ do
10. if $i < i \oplus \Sigma$ then
11. for each integer $j \in [0, Cntr_A[i])$ do
12. for each integer $k \in [0, Cntr_B[i \oplus \Sigma])$ do
13. $S_P = S_P \cup \{(P||HT_A[i][j], P'||HT_B[i \oplus \Sigma][k])\}$
Algorithm 17 Converting the Connectivity Problem into a SAT Problem

1: procedure SAT($M_1$, $M'_1$, $\alpha_1$)
2: \[ S_E = \emptyset \]
3: \[ A = f(M_1[0]), A' = f(M'_1[0]) \]
4: \[ A[828] = A[828] \oplus 1, A'[828] = A'[828] \oplus 1 \quad \triangleright \text{Padding} \]
5: \[ A[829] = A[829] \oplus 1, A'[829] = A'[829] \oplus 1 \quad \triangleright \text{Padding} \]
6: \[ A[830] = A[830] \oplus 1, A'[830] = A'[830] \oplus 1 \quad \triangleright \text{Padding} \]
7: for each integer $i \in [0, 828)$ do
8: \[ A[i] = v(i), A'[i] = v(i + 828) \]
9: \[ c = 1656 \]
10: for each integer $i \in [0, 320)$ do
11: \[ \Sigma[i] = v(c), \Sigma'[i] = v(c), \Sigma[\phi_1(i)] = v(c), \Sigma'[\phi_1(i)] = v(c) \]
12: \[ \Sigma[i] = v(c), \Sigma'[i] = v(c), \Sigma[\phi_2(i)] = v(c), \Sigma'[\phi_2(i)] = v(c) \]
13: \[ c = c + 1 \]
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: Add an XOR clause $v(c) + A'[i] + A'[i + 320] + A'[i + 640] + A'[i + 960] + A'[i + 1280] = 0$ to $S_E$
17: for each integer $i \in [0, 1600)$ do
18: \[ \rho(A), \pi(A), \rho(A'), \pi(A') \]
19: for each integer $i \in [0, 5)$ do
20: for each integer $j \in [0, 5)$ do
21: \[ B[i][j][k] = v(c) \quad \triangleright \text{Padding} \]
22: Add a clause $v(c) \lor \neg A[i][j + 1][k] \lor A[i][j + 1][k]$ to $S_E$
23: Add a clause $v(c) \lor \neg A'[i][j + 1][k]$ to $S_E$
24: Add a clause $v(c) \lor \neg A'[i][j + 1][k]$ to $S_E$
25: Add a clause $v(c) \lor \neg A'[i][j + 2][k]$ to $S_E$
26: Add a clause $v(c) \lor \neg A'[i][j + 2][k]$ to $S_E$
27: Add a clause $v(c) \lor \neg A'[i][j + 2][k]$ to $S_E$
28: Add an XOR clause $A[i] + A'[i] + B[i] + B'[i] = \alpha_1[i]$ to $S_E$
29: return $S_E$