Cryptanalysis of Subterranean-SAE

Fukang Liu[1,3], Takanori Isobe[2,3], Willi Meier[4]

[1] Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
liufukangs@163.com
[2] National Institute of Information and Communications Technology, Japan
[3] University of Hyogo, Hyogo, Japan
takanori.isobe@ai.u-hyogo.ac.jp
[4] FHNW, Windisch, Switzerland
willi.meier@fhnw.ch

Abstract. Subterranean 2.0 designed by Daemen, Massolino and Rotella is a Round 1 candidate of the NIST Lightweight Cryptography Standardization process. In the official document of Subterranean 2.0, the designers have made a cryptanalysis of the state collisions in unkeyed absorbing by reducing the number of rounds to absorb the message from 2 to 1. However, little cryptanalysis of the authenticated encryption scheme Subterranean-SAE is made. For Subterranean-SAE, the designers introduce 8 blank rounds to separate the controllable input and output, and expect that 8 blank rounds can achieve a sufficient diffusion. Therefore, it is meaningful to investigate the security by reducing the number of blank rounds. Moreover, the designers make no security claim but expect a non-trivial effort to achieve full-state recovery in a nonce-misuse scenario. In this paper, we present the first full-state recovery attack in a nonce-misuse scenario with practical time complexity 2^{16}. Moreover, in a nonce-respecting scenario and if the number of blank rounds is reduced to 4, we can mount a key-recovery attack with time complexity 2^{122} and data complexity $2^{69.5}$. The distinguishing attack can also be achieved with time and data complexity 2^{33}. Our cryptanalysis does not threaten the security claim for Subterranean-SAE and we hope it can enhance the understanding of Subterranean-SAE.

Keywords: AEAD, Subterranean 2.0, full-state recovery, distinguishing attack, key recovery, conditional cube tester

1 Introduction

The National Institute of Standards and Technology (NIST) started a public lightweight cryptography competition project in as early as 2013 and initiated the call for submissions in 2018, with the hope to select a lightweight cryptographic standard by combining the efforts of both academia and industry. The 56 Round 1 candidates of the NIST Lightweight Cryptography Standardization project became public on April 18, 2019. Such a competition is motivated by the development in many fields such as the sensor networks, healthcare, distributed control systems, and the Internet of Things, etc. With the development in these fields, some new requirements for a cryptographic primitive start to appear, covering the aspects of energy, power, area and throughput.

Although the competition is at the first round and there are so many primitives to be analyzed, it would be better to start the cryptanalysis as early as possible to help understand the security of the submitted candidates, which may in turn help determine which candidates should be moved to the next round for more attention. Since the publication of the submitted primitives, there has been a heated discussion at the lwc-forum Google Group[5] and the weaknesses of some constructions were pointed out. In addition, some weaknesses of the new proposed underlying primitives are also identified, like the very first probability 1 iterative differential characteristic in the SNEIK round function in [10], which was utilized to mount a forgery attack on full SNEIKEN in [5]. In addition, an iterative differential characteristic with probability 2^{-3} in TRIFLE-BC was also identified and used to mount an attack on reduced TRIFLE in [9].

Benefiting from the development in cryptanalysis in these years, some submitted primitives have been well analyzed by the designers. However, to have a better understanding, the third-party cryptanalysis is important as well. In this paper, our target is the primitive Subterranean 2.0 [2] designed by Daemen, Massolino and Rotella. The main reason is that we observed that its structures in keyed and unkeyed mode are interesting and its round function is very simple. As said by the designers in the official document [2], the round function is very simple and therefore it is an attractive target for cryptanalysis. Moreover, the degree of the one-round permutation is only 2, which gives us an impression that cube attack [3] and cube tester [1] may be feasible. In the recent three years, the cube attack has attracted the attention of many cryptographers. Especially, there is a series of publications of the application of cube attack to Keccak-based constructions [4,6,7,11]. Moreover, the bit-based division property introduced by Todo and Morii in [13] has also been applied to achieve a theoretical cube attack on stream ciphers in [12,14].

On the other hand, we observe that the designers of Subterranean 2.0 only investigated the security of state collisions in unkeyed absorbing by reducing the number of rounds to absorb the message from 2 to 1. However, there is little cryptanalysis for the authenticated encryption scheme Subterranean-SAE. We also noted in the official document [2] of Subterranean 2.0, that the designers made the following statement:

In nonce-misuse scenario or when unwrapping invalid cryptograms returns more information than a simple error, we make no security claims and an attacker may even be able to reconstruct the secret state. Nevertheless we believe that this would probably a non-trivial effort, both in attack complexity as in ingenuity.

Therefore, we are motivated to devise a full-state recovery attack in the nonce-misuse scenario. In addition, the blank rounds in Subterranean-SAE are used to separate the controllable input and output and the designers choose 8 blank rounds. Thus, we believe that it is still interesting and meaningful to investigate its security when the number of blank rounds is reduced.

[5] `https://groups.google.com/a/list.nist.gov/forum/#!forum/lwc-forum`

Our Contributions. Inspired from the idea of the conditional cube tester proposed by Huang et al. [4], we propose four types of conditional cube tester, each of which requires that the number of the conditions involving the secret bits is 1. As far as we know, the additional constraint on the number of conditions is new, which is not well studied in previous work, although such a case has appeared in [7]. Then, we can mount three types of attacks on Subterranean-SAE as follows. Our results [6] are summarized in Table 1.

- For the full-state recovery attack in a nonce-misuse scenario, the cube size is rather small, i.e. 2 or 3. Therefore, we have to carefully trace the propagation of the cube variables so that we can construct a deterministic and strict distinguisher. Inspired from the idea of [8], we can manage to detect the propagation of the cube variables in a dedicated way and finally determine valid cube variables of small cube size. With such four conditional cube testers, some secret state bits can be recovered. Finally, we guess some extra secret state bits to construct sufficient linear equations. In this way, the full-state recovery attack on Subterranean-SAE can be achieved with practical time complexity 2^{16}.
- With a similar dedicated tracing method, we found 33-dimensional cube variables which can be used to mount a distinguishing attack on Subterranean-SAE if the number of blank rounds is reduced to 4.
- When the number of blank rounds is reduced to 4, the key-recovery attack is also feasible. The attack procedure is composed of two steps. The first step is to recover some secret state bits as in the full-state recovery attack. The second step is to guess some key bits to construct a linear boolean equations system, each solution of which corresponds to the full key. In this way, we can achieve the key-recovery attack with time complexity 2^{122} and data complexity $2^{69.5}$.

This paper is organized as follows. We briefly introduce Subterranean 2.0 and the cube attack, cube tester and conditional cube tester in Section 2. Then, the full-state recovery attack is described in Section 3. The distinguishing attack and key-recovery attack are shown in Section 4 and Section 5, respectively. Finally, the paper is concluded in Section 6.

Table 1: The analytical results of Subterranean-SAE

Attack Type	Blank rounds	Data	Time	Nonce-misuse	Ref.
Full-state recovery attack	arbitrary	1177	2^{16}	Yes	Section 3
Distinguishing attack	4/8	2^{33}	2^{33}	No	Section 4
Key-recovery attack	4/8	$2^{69.5}$	2^{122}	No	Section 5

[Data] This represents the number of messages. The length of each message in the query is not greater than $32 \times 7 = 224$ bits.
[Time] This represents the required number of encryption queries.

[6] The source code to verify how to recover the secret state bits and the distinguishing attack is available at `https://github.com/Crypt-CNS/Subterranean-SAE.git`

2 Preliminaries

In this section, we will give an introduction of the round function of Subterranean 2.0 and its authenticated encryption scheme Subterranean-SAE. More details can be referred to the official document [2]. Moreover, since our technique benefits from the development of cube attack, we will also briefly describe the main idea of cube attack [3], cube tester [1] and conditional cube tester [4].

2.1 Description of Subterranean 2.0

The subterranean 2.0 round function is composed of 4 simple operations and operates on a 257-bit state. Denote the 257-bit state by s and the four operations by χ, ι, θ, π. The one-round permutation $R = \pi \circ \theta \circ \iota \circ \chi$ is detailed as follows, where $s[i]$ represents the i-th bit of s.

$$\begin{aligned}
\chi &: s[i] \leftarrow s[i] \oplus \overline{s[i+1]}s[i+2], \\
\iota &: s[0] \leftarrow s[0] \oplus 1, \\
\theta &: s[i] \leftarrow s[i] \oplus s[i+3] \oplus s[i+8], \\
\pi &: s[i] \leftarrow s[12i],
\end{aligned}$$

where $0 \leq i \leq 256$. In addition, we denote the state after χ, ι, θ operation by s_χ, s_ι and s_θ, respectively.

2.2 The Subterranean-SAE Authenticated Encryption Scheme

Based on the subterranean 2.0 round function, the designers have constructed an authenticated encryption scheme named Subterranean-SAE, as illustrated in Figure 1. In this scheme, the input consists of a 128-bit key K, a 128-bit nonce N, an associated data A and a message M. The output is the ciphertext C and tag T. The procedure to generate the ciphertext and tag can be briefly described as follows:

Step 1: **Absorb the key:** Initialize a state s with all bits set to 0. Split the 128-bit key K into four 32-bit blocks K_0, K_1, K_2 and K_3. Then, make four times of consecutive calls of $duplex(s, K_i)$ $(0 \leq i \leq 3)$ to update the internal state. Finally, make a call of $duplex(s, NULL)$ to further update the internal state, where $NULL$ represents an empty string.

Step 2: **Absorb the nonce:** Split the 128-bit nonce N into four 32-bit blocks N_0, N_1, N_2 and N_3. Then, make four times of consecutive calls of $duplex(s, N_i)$ $(0 \leq i \leq 3)$ to update the internal state. Finally, make a call of $duplex(s, NULL)$ to further update the internal state.

Step 3: **Blank rounds:** Make 8 times of consecutive calls of $duplex(s, NULL)$ to update the internal state.

Step 4: **Absorb the associated data:** Split the $|A|$-bit associated data A into a series of 32-bit blocks, denoted by A_i $(0 \leq i < \lceil |A|/32 \rceil)$, where $|A|$ denotes the length of A. Then, make $\lceil |A|/32 \rceil$ times of consecutive calls of $duplex(s, A_i)$ $(0 \leq i < \lceil |A|/32 \rceil)$ to update the internal state. If $|A|$ is a multiple of 32 (the case when A is empty also belongs to this case), make one more call of $duplex(s, NULL)$ to update the internal state.

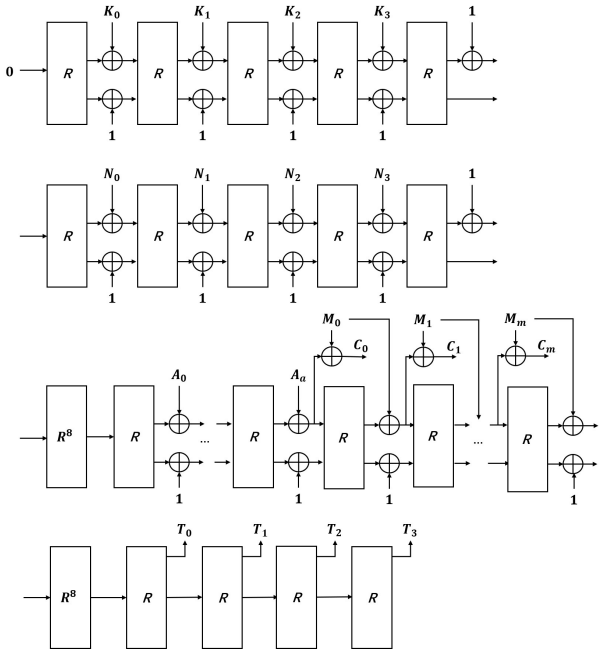

Fig. 1: The construction of Subterranean-SAE

Step 5: **Message encryption:** Split the $|M|$-bit ($|M| \geq 0$) message M into a series of 32-bit blocks, denoted by M_i ($0 \leq i < \lceil |M|/32 \rceil$), where $|M|$ denotes the length of M. Then, make $\lceil |M|/32 \rceil$ times of consecutive calls of $duplex(s, M_i)$ ($0 \leq i \leq \lceil |M|/32 \rceil$) to update the internal state. Before each call of $duplex(s, M_i)$, make a call of $extract(s)$ ($0 \leq i < \lceil |M|/32 \rceil$) and then the corresponding ciphertext is $C_i = extract(s) \oplus M_i$. If $|M|$ is a multiple of 32 (the case when M is empty also belongs to this case), make one more call of $duplex(s, NULL)$ to update the internal state.

Step 6: **Blank rounds:** Make 8 times of consecutive calls of $duplex(s, NULL)$ to update the internal state.

Step 7: **Extract tag:** Make 4 times of consecutive calls of $duplex(s, NULL)$. After each call of $duplex(s, NULL)$, make a call of $extract(s)$ to obtain 32-bit T_i ($0 \leq i \leq 3$).

The details of $duplex(s, \sigma)$ and $extract(s)$ are described in Algorithm 1 and Algorithm 2, where σ is a bit string with at most 32 bits. The readers can also refer to the official document of Subterranean 2.0 [2] for a better understanding.

The pseudocode in Algorithm 1 and Algorithm 2 is slightly different from the official document since we introduced two extra arrays order0[] and order1[]. The details of the order0[] and order1[] are specified in Table 2.

Algorithm 1 *duplex*(s, σ)

1: $R(s)$
2: $j = 0$
3: **for** j from 0 to $|\sigma| - 1$ **do**
4: s[order0[j]]=s[order0[j]]$\oplus\sigma$[j]
5: **end for**
6: s[order0[j]]=s[order0[j]]$\oplus$1

Algorithm 2 *extract*(s)

1: **for** j from 0 to 31 **do**
2: z[j]=s[order0[j]]$\oplus$ s[order1[j]]
3: **end for**
4: return z

2.3 Cube Tester

Cube tester was first proposed by Aumasson et al. at FSE 2009 [1] after Dinur et al. introduced cube attack at Eurocrypt 2009 [3]. Different from standard cube attack, which aims at key extraction, cube tester performs non-randomness detection. In our paper, we only concentrate on a specific non-random behaviour, i.e. the cube sum is zero. To describe cube tester, we first recall the concept of cube attack as follows.

Theorem 1. [3] *Given a polynomial* $F : \{0,1\}^n \rightarrow \{0,1\}$ *of degree d. Suppose* $0 < k < d$ *and t denotes the monomial* $x_0...x_{k-1}$*. Then, F can be written as*

$$F = t \cdot P_t(x_k, \ldots, x_{n-1}) + Q_t(X),$$

where none of the terms of $Q_t(X)$ *is divisible by t. Then the sum of F over all values of the cube (defined by t) is*

$$\sum_{x' \in CU_t} F = \sum_{x' \in CU_t} F(x', x_k, \ldots, x_{n-1}) = P_t(x_k, \ldots, x_{n-1}).$$

Table 2: The details of order0[] and order1[]

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
order0[i]	1	176	136	35	249	134	197	234	64	213	223	184	2	95	15	70	241
i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	–
order0[i]	11	137	211	128	169	189	111	4	190	30	140	225	22	17	165	256	–
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
order1[i]	256	81	121	222	8	123	60	23	193	44	34	73	255	162	242	187	16
i	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	–	–
order1[i]	246	120	46	129	88	68	146	253	67	227	117	32	235	240	92	–	–

If there exists such a cube CU_t that the following equation always hold, then CU_t can be viewed as one type of cube tester [1], i.e. the sum over it always equals zero.

$$\sum_{x' \in CU_t} F = \sum_{x' \in CU_t} F(x', x_k, \ldots, x_{n-1}) = P_t(x_k, \ldots, x_{n-1}) = 0.$$

For example, consider the following polynomial F:

$$F(x_0, x_1, x_2, x_3) = x_0x_1 + x_1x_2 + x_2x_4 + x_1x_3 + x_1x_2x_4.$$

Then, the following equation always holds:

$$\sum_{(x_0,x_3)\in\{0,1\}^2} F(x_0, x_1, x_2, x_3) = 0.$$

The reason is that that none of the monomial in $F(x_0, x_1, x_2, x_3)$ is divisible by x_0x_3. However, if we sum F over all values of (x_1, x_2), then we can obtain the following equation:

$$\sum_{(x_1,x_2)\in\{0,1\}^2} F(x_0, x_1, x_2, x_3) = 1 + x_4.$$

That is, the sum is dependent on the value of x_4.

2.4 Conditional Cube Tester

Conditional cube tester was first proposed by Huang et al. [4], which was used to detect the non-randomness of Keccak-based constructions, i.e. the cube sum is zero. Different from the standard cube tester, conditional cube tester works only when certain conditions hold. For example, consider the following polynomial F, where c is an unknown variable over GF(2).

$$F(x_0, x_1, x_2, x_3) = c \cdot x_0x_1 + x_1x_2 + x_2x_4 + x_1x_3 + x_1x_2x_4.$$

If we have some conditions to ensure that $c = 0$ always holds, then

$$\sum_{(x_0,x_1)\in\{0,1\}^2} F(x_0, x_1, x_2, x_3) = 0.$$

However, when c can not be controlled and is randomly chosen, then the sum of F over all values of (x_0, x_1) can not be predicted and behaves randomly as well.

2.5 Our Conditional Cube Tester

According to the general definition of conditional cube tester, once the attacker determines the cube variables and the corresponding conditions involving the secret information, he can manage to extract the secret information as well. Specifically, if the cube sum is zero, he can make a decent conclusion that the conditions hold

simultaneously, thus collecting equations of the involved secret information. However, such a conclusion may be wrong since it is still possible that the cube sum will be zero even if the conditions are not satisfied. Hence, there are two directions to confirm the correctness of such a conclusion. One direction is to obtain a strict distinguisher which can distinguish two cases (cube sum is zero and nonzero) with success probability 1. The second direction is to introduce sufficient number of cube variables and number of rounds to diffuse them, which will make the cube sum unpredictable if the conditions do not hold, i.e. the event that the cube sum is zero in this case happens with an overwhelming probability. Our paper will consider both directions.

3 Full-State Recovery Attack

In this section, we will describe how to mount a full-state recovery attack for Subterranean-SAE in a nonce-misuse scenario.

3.1 Overview

Suppose the same nonce can be reused for several times. Our attack procedure can be divided into two steps.

Step 1: Use four types of conditional cube tester to recover some secret state bits.
Step 2: Guess extra unknown secret state bits to construct a sufficient number of linear boolean equations. Solve the equation system to recover the full state and check its correctness according to the ciphertext and tag value.

Since the Step 1 is to recover the secret state bits using a conditional cube tester, we first briefly introduce its general idea and procedure. The conditional cube tester [4] can be viewed as a distinguisher. Only when certain linear conditions hold will the cube sum be zero. Therefore, the attacker will try to find a way to involve the secret bits into these conditions. Once he determines the conditions and the corresponding cube variables, he can continuously adjust the values of the public bits involved in the linear conditions until he observed that the cube sum becomes zero. After repeating the above procedure with different cube variables, a certain number of linear equations in terms of secret bits can be collected. Finally, solve the equation system to recover the secret bits.

For Step 2, after the attacker guesses some extra secret state bits, he can collect a sufficient number of linear equations. Then, he can solve this linear equation system with the well-known Gauss elimination algorithm to obtain one solution. For each solution, the full state is recovered and the attacker can verify its correctness by computing the tag and the ciphertext and comparing them with the correct value. A match will suggest the correct value of the secret state.

Additional Constraint. In recent two years, the conditional cube tester has been intensively investigated [4,6,7,8,11]. However, in order to apply it to Subterranean-SAE, additional constraint to build the conditional cube tester is essential, which has

not been clearly discussed before, as far as we know. The additional constraint is the number of conditions involving the secret bits, as specified below.

Additional constraint: The number of conditions involving the secret bits is one. Denote this condition by L(x) = 0, where L(x) represents a linear expression of x. It can thus be concluded that L(x)=1 if the cube sum is nonzero. The conclusion that L(x) = 0 if the cube sum is zero requires strictly random behavior in order that L(x) = 0 can be concluded.

What benefits will this additional constraint bring? Once we observe that the cube sum is not zero, we can immediately obtain an equation $L(x) = 1$. Once the cube sum is zero, we can also construct an equation $L(x) = 0$. In other words, whatever the cube sum is, there is no need to adjust the values of public bits in $L(x)$ and and we can directly collect an equation.

However, once there are more than 1 conditions involving the secret bits, suppose they are $L_0(x) = 0$ and $L_1(x) = 0$. When the cube sum is zero, two linear equations $L_0(x) = 0$ and $L_1(x) = 0$ are obtained. However, when the cube sum is not zero, there will be three possible values for $(L_0(x), L_1(x))$, which are $(0, 1)$, $(1, 0)$ and $(1, 1)$. If the attacker has no control of the bits in the expressions $L_0(x)$ and $L_1(x)$, then he obviously cannot know the actual value of $L_0(x)$ or $L_1(x)$. As far as we know, in all previous applications of conditional cube tester to Keccak-based constructions, the attacker can always have control of the bits involved into the key-dependent bit conditions, which is the public message. Thus, he can always adjust the message until he observes that the cube sum becomes zero.

As will be shown, to apply the conditional cube tester to Subterranean-SAE, the condition is directly imposed on one secret state bit. Thus, once more bit conditions are involved, the attacker cannot determine which condition holds if the cube sum is not zero. In this case, he can only know that the conditions do not hold simultaneously.

3.2 Determining Parameters for the Conditional Cube Tester

Since the publication of conditional cube tester [4], MILP-based methods to search the corresponding parameters have been developed [6,11]. In addition, there is also a dedicated method to search the cube variables for Keccak-MAC in [8]. Our way to search the parameters for conditional cube tester is based on a similar idea of the dedicated method [8].

There are four types of conditional cube tester in our attack, denoted by TYPE-I, TYPE-II, TYPE-III and TYPE-IV conditional cube tester respectively. The method to determine the parameters will vary for different types of conditional cube tester.

To make this part understandable, we first give an illustration of how the message is processed in Subterranean-SAE, as shown in Figure 2. The input and output of the round permutation is denoted by MS_i^{in} and MS_i^{ot} when processing the message block, i.e. $MS_i^{ot} = R(MS_i^{in})$.

Before the attack, we send an encryption query $(N, A, M_0, M_1, M_2, M_3)$ to collect the corresponding tag T_0, where $M_i = 0$ $(0 \leq i \leq 2)$ for simplicity. Our first aim is to recover some bits of the secret states $(MS_1^{in}, MS_2^{in}, MS_3^{in})$ and the final aim is to recover the full MS_1^{in} in this query.

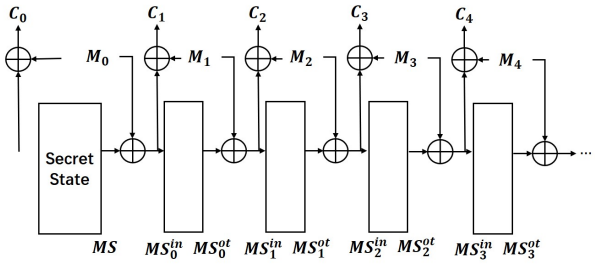

Fig. 2: Processing the message

According to the process to absorb the message in Figure 2, it can be observed that the attacker can always control 32 bits of the input to the round permutation as well as extract 32-bit information after one-round permutation. Thus, we use an equivalent description of this process, as depicted in Figure 3. Specifically, s^i ($i \geq 0$) denotes the input of the $(i+1)$-th round permutation, while s^i_χ, s^i_ι, s^i_θ denotes the state after χ, ι, θ operation in the $(i+1)$-th round, respectively. Note that the attacker can always control 32 bits of s^i and extract 32-bit information of s^i by making a call of $z^i = extract(s^i)$.

In the following part, we suppose the secret input states are (s^0, s^1) and will introduce how to use the four types of conditional cube tester to recover some of their secret bits. Then, we will describe how to deploy it to recover some secret bits of $(MS^{in}_1, MS^{in}_2, MS^{in}_3)$.

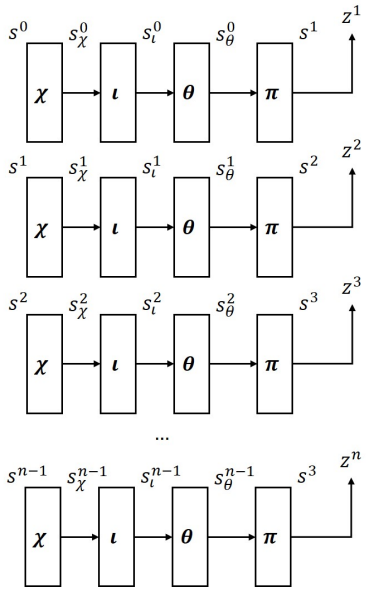

Fig. 3: Equivalent description of processing the message

TYPE-I Conditional Cube Tester. For the TYPE-I condition cube tester, we only choose two cube variables v_0 and v_1, where v_0 and v_1 are set at s^0 and s^1 respectively. The condition is imposed on a certain bit $s^0[x]$, denoted by $f(s^0[x]) = 0$, where $f(s^0[x])$ is either $s^0[x]$ or $s^0[x] \oplus 1$. Then, v_0 and v_1 should have the following constraints:

Constraint 1: If $f(s^0[x]) = 0$ holds, after one-round permutation for v_0, v_0 will not be next[7] to v_1. In this case, z^2 is linear in (v_1, v_2).

Constraint 2: If $f(s^0[x]) = 0$ does not hold, after one-round permutation for v_0, v_0 will be next to v_1 and some bits of z^2 will contain the quadratic term v_1v_2 with probability 1.

An illustration for the TYPE-I condition cube tester is given in Figure 4.

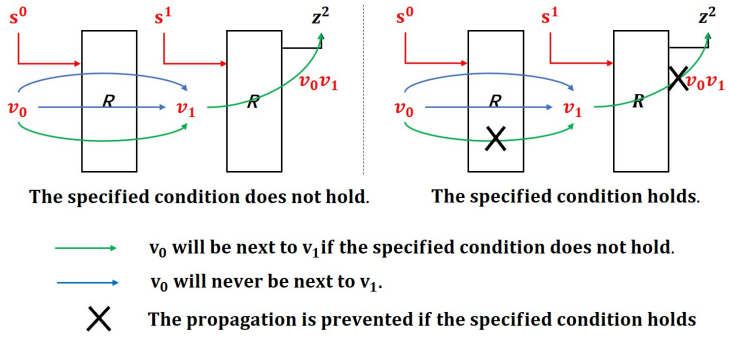

Fig. 4: Illustration of TYPE-I conditional cube tester

TYPE-II Conditional Cube Tester. For the TYPE-II condition cube tester, we choose three cube variables v_0, v_1 and v_2, where v_0 and (v_1, v_2) are set at s^0 and s^1 respectively. The condition is imposed on a certain bit $s^0[x]$, denoted by $f(s^0[x]) = 0$, where $f(s^0[x])$ is either $s^0[x]$ or $s^0[x] \oplus 1$. Then, v_0 and (v_1, v_2) should have the following constraints:

Constraint 1: v_1 and v_2 are not next to each other, i.e. they will not multiply with each other after one round permutation.

Constraint 2: If $f(s^0[x]) = 0$ holds, after one-round permutation for v_0, v_0 will not be next to v_1 nor v_2. In this case, z^2 is linear in (v_0, v_1, v_2). Since the degree of the one-round permutation is 2, z^3 will not contain the term $v_0v_1v_2$.

Constraint 3: If $f(s^0[x]) = 0$ does not hold, after one-round permutation for v_0, v_0 will be next to v_1. In addition, after one more round permutation, z^3 will contain the cubic term $v_0v_1v_2$ with probability 1.

An illustration for the TYPE-II condition cube tester is given in Figure 5.

[7] 'Next' here means that v_0 and v_1 have adjacent indices as state bits. Same meaning for the remaining part of this paper.

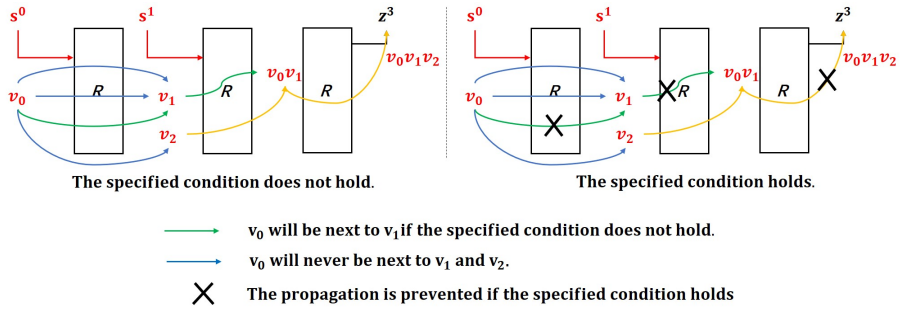

Fig. 5: Illustration of TYPE-II conditional cube tester

TYPE-III Conditional Cube Tester. For the TYPE-III condition cube tester, we choose two cube variables v_0 and v_1, where v_0 and v_1 are set at s^0 and s^2 respectively. The condition is imposed on a certain bit $s^0[x]$, denoted by $f(s^0[x]) = 0$, where $f(s^0[x])$ is either $s^0[x]$ or $s^0[x] \oplus 1$. Then, v_0 and v_1 should have the following constraints:

Constraint 1: If $f(s^0[x]) = 0$ holds, after two-round permutation for v_0, v_0 will not be next to v_1. In this case, z_3 will not contain the term v_0v_1.

Constraint 2: If $f(s^0[x]) = 0$ does not hold, after two-round permutation for v_0, v_0 will be next to v_1 and some bits of z_3 will contain the term v_0v_1 with probability 1.

An illustration for the TYPE-III condition cube tester is given in Figure 6.

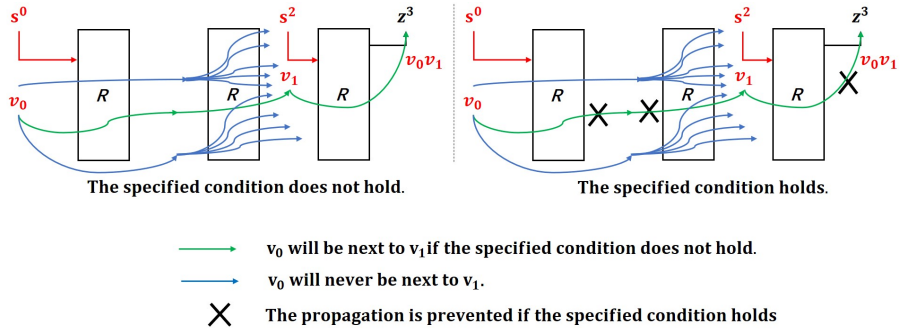

Fig. 6: Illustration of TYPE-III conditional cube tester

TYPE-IV Conditional Cube Tester. For the TYPE-IV condition cube tester, we choose two cube variables v_0 and v_1, where v_0 and v_1 are set at s^0 and s^2 respectively. Different from the previous three types of conditional cube tester, the condition is

imposed on a certain bit $s^1[x]$ rather than $s^0[x]$, denoted by $f(s^1[x]) = 0$, where $f(s^1[x])$ is either $s^1[x]$ or $s^1[x] \oplus 1$. Then, v_0 and v_1 should have the following constraints:

Constraint 1: If $f(s^1[x]) = 0$ holds, after two-round permutation for v_0, v_0 will not be next to v_1. In this case, z_3 will not contain the term v_0v_1.
Constraint 2: If $f(s^1[x]) = 0$ does not hold, after two-round permutation for v_0, v_0 will be next to v_1 and some bits of z_3 will contain the term v_0v_1 with probability 1.
Constraint 3: The value of $f(s^1[x])$ will remain the same if v_0 takes different values.

The TYPE-IV conditional cube tester will allow us to recover more secret state bits. An illustration for the TYPE-IV condition cube tester is given in Figure 7. One can easily capture the difference between TYPE-III and TYPE-IV condition cube tester according to the illustrations.

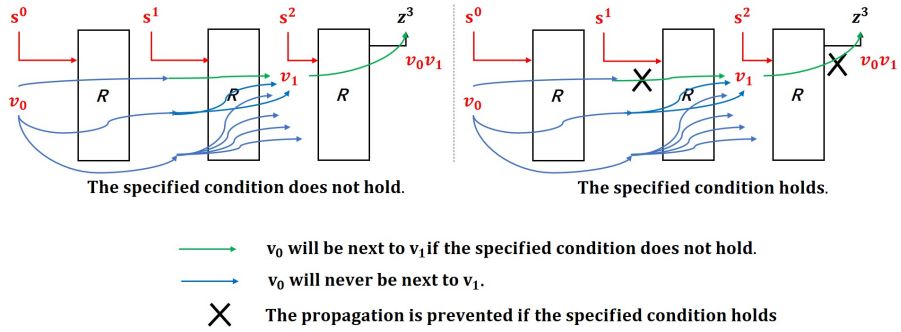

Fig. 7: Illustration of TYPE-IV conditional cube tester

Tracing Propagation of Cube Variables. Suppose a variable v is set at the input state bit $s^i[p]$. According to the definition of χ operation, there will be three bits containing the variable. We classify the bits into three types in a similar way as [8] in order to achieve better tracing.

- **Core bit**: The bit $s^i_\chi[p]$ is defined as the core bit since $s^i_\chi[p]$ will always contain the variable v. After ι, θ and π operations, it will propagate to three bit positions of s^{i+1}, which will be stored in the array CORE[] of size 3 ($0 \le i \le 2$).
- **Zero-condition bit**: The bit $s^i_\chi[p-1]$ is defined as the zero-condition bit since $s^i_\chi[p-1]$ will not contain the variable v if $s^i[p+1] = 0$. The variable v in $s^i_\chi[p-1]$ will propagate to three bit positions of s^{i+1}, which will be stored in the array ZERO[] of size 3.
- **One-condition bit**: The bit $s^i_\chi[p-2]$ is defined as the one-condition bit since $s^i_\chi[p-2]$ will not contain the variable v if $s^i[p-1] = 1$. The variable v in $s^i_\chi[p-2]$ will propagate to three bit positions of s^{i+1}, which will be stored in the array ONE[] of size 3.

Searching Cube Variables for TYPE-I Conditional Cube Tester. Suppose v_0 is set at $s^0[k]$ ($k \in \{e|e = \text{order0}[j], 0 \le j \le 31\}$). Using the above tracing algorithm, we can obtain CORE[i], ZERO[i] and ONE[i] ($0 \le i \le 3$) for the propagation of v_0. Then we determine the compatible cube variable v_1 set at s^1 according to CORE[i], ZERO[i] and ONE[i] ($0 \le i \le 3$) with Algorithm 3 in Appendix A. According to the result obtained from Algorithm 3, it is not sufficient to determine whether a candidate for v_1 is valid. The reason is that the attacker can only extract fixed 32-bit information z^2 of s^2, where $z^2[i] = s[\text{order0}[i]] \oplus s[\text{order1}[i]]$ ($0 \le i \le 31$). Suppose the condition used to slow down the propagation of v_1 does not hold, then s^1_χ will contain the term v_0v_1. However, the attacker cannot ensure the quadratic term v_0v_1 will propagate to z^2. In this case, z^2 is still linear in (v_0, v_1). Therefore, for each candidate of v_1, we have to make a further filtering. Suppose v_1 is set at one candidate bit position q and the zero-bit/one-bit bit of v_0 will propagate to $s^1[q-1]$ (or $s^1[q+1]$). Then, we trace the propagation of a variable v_q (which is quadratic) set in $s^1[q-2]$ (or $s^1[q-1]$) with the above tracing method and obtain the array CORE$'$. At last, we check whether there is an element CORE[i] ($0 \le i \le 2$) satisfying

$$\text{CORE}'[i] = \text{order0}[j], \text{CORE}'[i] \neq \text{order1}[j],$$

or

$$\text{CORE}'[i] \neq \text{order0}[j], \text{CORE}'[i] = \text{order1}[j],$$

where $0 \le j \le 31$. If there is, the candidate bit position q is valid. In other words, if the predefined bit condition $f(s^0[x]) = 0$ does not hold, at least one bit of z^2 will contain the quadratic term v_0v_1. If it holds, z^2 is linear in (v_0, v_1). Thus, we can obtain an equation based on the cube sum of z^2 as follows:

$$\sum z^2 \neq 0 \Rightarrow f(s^0[x]) = 1,$$
$$\sum z^2 = 0 \Rightarrow f(s^0[x]) = 0,$$

where $f(s^0[x])$ is either $s^0[x]$ or $s^0[x] \oplus 1$.

With this method to select cube variables, we can find 24 possible choices for (v_0, v_1) and recover 24 secret bits of s^0, as listed in Table 3. To have a better understanding of this table, we give an explanation for one choice. Consider the parameter that v_0 is set at $s^0[2]$ and v_1 is set at $s^1[213]$. If the condition $s^0[3] = 0$ does not hold, the cube sum of z^2 will be nonzero. Therefore, if we observe that the cube sum of z^2 is zero, we can know that $s^0[3] = 0$. Otherwise, $s^0[3] = 1$.

Searching Cube Variables for TYPE-II Conditional Cube Tester. For TYPE-II conditional cube tester, the number of cube variables is 3. Therefore, only when the secret bits cannot be recovered by TYPE-I, TYPE-III and TYPE-IV conditional cube tester will we use it. Note that v_0 is set at s^0. Similarly, we will obtain the candidate for v_1 set at s^1. However, different from TYPE-I conditional cube tester, we do not filter the case when v_0v_1 does not appear at z^2. We still trace the propagation of the quadratic term v_0v_1 to the state s^2 and record the bit positions in s^2 which always contain the

Table 3: Parameters for TYPE-I conditional cube tester

Position of v_0	2	4	11	15	22	64	64	70	95	95	111	128
Position of v_1	213	22	128	128	2	197	111	176	30	137	136	95
Position of condition	3	5	10	16	21	65	63	69	96	94	112	129
Value of condition	0	0	1	0	1	0	1	1	0	1	0	0
Position of v_0	128	134	136	165	169	197	197	211	213	225	234	241
Position of v_1	140	95	140	184	184	165	17	211	190	189	189	190
Position of condition	127	133	135	166	168	198	196	212	214	226	233	240
Value of condition	1	1	1	0	1	0	1	0	0	0	1	1

term v_0v_1. In addition, we also trace the propagation of v_2 set in s^1 to the state s^2 and record the bit positions in s^2 which always contain the term v_2. We expect that after χ operation, there will always exist a cubic term $v_0v_1v_2$ in s^2_χ, which can be easily detected with the recorded bit positions for the propagation of v_0v_1 and v_2. Moreover, the cubic term will also always propagate to the output bits in z^3. Therefore, we can construct a distinguisher as follows:

If the predefined bit condition $f(s^0[x]) = 0$ does not hold, at least one bit of z^3 will contain the cubic term $v_0v_1v_2$. If it holds, s^2 is linear in (v_0, v_1, v_2) and z^3 will obviously not contain the cubic term $v_0v_1v_2$. Thus, we can obtain an equation based on the cube sum of z^3 as follows:

$$\sum z^3 \neq 0 \Rightarrow f(s^0[x]) = 1,$$
$$\sum z^3 = 0 \Rightarrow f(s^0[x]) = 0,$$

where $f(s^0[x])$ is either $s^0[x]$ or $s^0[x] \oplus 1$.

In this case, we have 2 choices for (v_0, v_1, v_2) and can recover 2 secret bits of s^0, as listed in Table 4. The explanation for this table is the same as that for Table 3.

Table 4: Parameters for TYPE-II conditional cube tester

Position of v_0	1	2
Position of (v_1, v_2)	(1,11)	(1,11)
Position of condition	2	1
Value of condition	0	1

Searching Cube Variables for TYPE-III Conditional Cube Tester. For the TYPE-III conditional cube tester, the number of cube variables is two. The cube variables (v_0, v_1) are set at s^0 and s^2 respectively. Similarly, we first obtain three kinds of bit

positions in s^1 which will contain the variable v_0 and record them in the array CORE, ZERO and ONE respectively. Next, for the three bit positions in CORE, we trace their propagation to s^2 and record all the possible influenced bit positions in the array CORE_2. Similarly, we can obtain ZERO_2 and ONE_2 to record the possible influenced bit positions of s^2 caused by the variables in the bit positions ZERO and ONE in s^1. Note that ZERO_2 and ONE_2 will contain all possible influenced bit positions. However, the information provided by CORE_2, ZERO_2 and ONE_2 is still not sufficient to help determine a candidate position for v_1. The reason is that some bits of s^1 will influence the propagation of the variables located at ZERO and ONE. Therefore, to find out which bits of s^2 will always contain the variable propagating from the bit positions ZERO (or ONE) of s^1, we compute two extra arrays ZEROCore and ONECore. ZEROCore/ONECore is used to record the bit positions of s^2 that always contain the variable if there is a variable in the bit positions ZERO/ONE of s^1. Based on the five arrays $\text{CORE}_2[]$, $\text{ZERO}_2[]$, $\text{ONE}_2[]$, ZEROCore[] and ONECore[], we define additional five arrays $\text{core}_2[]$, $\text{zero}_2[]$, $\text{one}_2[]$, zeroCore[] and oneCore[], which are used to record which bits in order0[] (only the first 32 elements) will be next to the element in $\text{CORE}_2[]$, $\text{ZERO}_2[]$, $\text{ONE}_2[]$, ZEROCore[] and ONECore[] respectively. For example, supposing $\text{CORE}_2[0]$=164 or $\text{CORE}_2[0]$=166, we will add 165 to the array $\text{core}_2[]$ since order0[31]=165.

At last, we can determine a candidate bit position in s^2 for v_1. The bit position q ($q \in \{e|e = \text{order0}[j], 0 \leq j \leq 31\}$) can be viewed as a candidate only if it satisfies the following condition:

$$q \in \text{zeroCore}, q \notin \text{core}_2, q \notin \text{one}_2,$$

or

$$q \in \text{oneCore}, q \notin \text{core}_2, q \notin \text{zero}_2.$$

A valid bit position p for v_1 should satisfy one more condition. For example, suppose $p - 1 \in$ ZEROCore (or $p + 1 \in$ ZEROCore). Then at lease one bit of z^3 will contain the term $s^2[p]s^2[p-1]$ (or $s^2[p]s^2[p+1]$) with probability 1. In other words, if the propagation of v_0 is not prevented by a condition, a quadratic term v_0v_1 will always appear at the expression of s^3. However, if such a propagation is prevented, s^3 is linear in (v_0, v_1). Therefore, we can construct a distinguisher as follows:

If the predefined bit condition $f(s^0[x]) = 0$ does not hold, at least one bit of z^3 will contain the cubic term v_0v_1. If it holds, s^2 is linear in (v_0, v_1) and z^3 will obviously not contain the cubic term v_0v_1. Thus, we can obtain an equation based on the cube sum of z^3 as follows:

$$\sum z^3 \neq 0 \Rightarrow f(s^0[x]) = 1,$$
$$\sum z^3 = 0 \Rightarrow f(s^0[x]) = 0,$$

where $f(s^0[x])$ is either $s^0[x]$ or $s^0[x] \oplus 1$.

In total, we have 27 possible choices for (v_0, v_1) can recover 27 secret bits of s^0, as listed in Table 5. The explanation for this table is the same as that for Table 3.

Table 5: Parameters for TYPE-III conditional cube tester

Position of v_0	1	11	15	17	22	30	30	35	35	70	111	136	137	140
Position of v_1	15	111	35	35	35	197	11	1	11	140	35	1	1	223
Position of condition	0	12	14	18	23	31	29	36	34	71	110	137	136	141
Value of condition	1	0	1	0	0	0	1	0	1	0	1	0	1	0

Position of v_0	140	165	169	176	176	184	190	211	223	234	241	249	249	–
Position v_1	169	11	30	95	211	2	11	70	189	22	2	95	2	–
Position of condition	139	164	170	177	175	185	191	210	224	235	242	248	250	–
Value of condition	1	1	0	0	1	0	0	1	0	0	0	1	0	–

Searching Cube Variables for TYPE-IV Conditional Cube Tester. For the TYPE-IV conditional cube tester, the number of cube variables is two. The cube variables (v_0, v_1) are set at s^0 and s^2 respectively. Similarly, we first obtain three kinds of bit positions in s^1 which will contain the variable v_0 and record them in the array CORE, ZERO and ONE respectively. Next, for the three bit positions in CORE, we trace their propagation one by one and record all the possible bit positions of s^2 that will contain the variables propagating from CORE[0], CORE[1] and CORE[2] of s^1 in CORE0, CORE1 and CORE2, respectively. Then, we further classify the positions of CORE0, CORE1 and CORE2. Taking the propagation of CORE[0] as instance. According to the tracing method, the positions of s^2 containing the variables propagating from CORE[0] of s^1 can be classified into three types and we denote them by CORE0core, CORE0zero, CORE0one. CORE0core stores the positions that always contain the variables propagating from CORE[0] of s^1. CORE0zero stores the positions that contain the variables propagating from CORE[0] of s^1 based on a bit condition on s^1 whose value is zero. CORE0one stores the positions that contain the variables propagating from CORE[0] of s^1 based on a bit condition on s^1 whose value is one. Similarly, we can obtain CORE1core, CORE1zero, CORE1one, CORE2core, CORE2zero and CORE2one.

Now, we deal with ZERO and CORE. For both of them, we record all the possible bit positions of s^2 containing the variables propagating from ZERO and ONE of s^1 in ZEROAll and ONEAll respectively.

Finally, similar to the the search for TYPE-III conditional cube cube variables, we additionally define 14 arrays CORE0Next, CORE1Next, CORE2Next, CORE0coreNext, CORE0zeroNext, CORE0oneNext, CORE1coreNext, CORE1zeroNext, CORE1oneNext, CORE2coreNext, CORE2eroNext, CORE2oneNext, ZEROALLNext and ONEAllNext to record which bits in order0[] (only the first 32 elements) will be next to the element in CORE0, CORE1, CORE2, CORE0core, CORE0zero, CORE0one, CORE1core, CORE1zero, CORE1one, CORE2core, CORE2ero, CORE2one, ZEROALL and ONEAll, respectively. For example, supposing CORE0[0]=164 or CORE0[0]=166, we will add 165 to the array CORE0Next since order0[31]=165.

Based on the newly defined 14 arrays, we can determine a candidate position denoted by p for v_1 in s^2. The value of p has to satisfy one of the follow 6 conditions:

Condition 1:

$$\begin{cases} p \in \text{CORE0zeroNext} \\ p \notin \text{CORE0oneNext} \\ p \notin \text{CORE0coreNext} \\ p \notin \text{CORE1Next} \\ p \notin \text{CORE2Next} \\ p \notin \text{ZEROAllNext} \\ p \notin \text{ONEAllNext} \end{cases}$$

Condition 2:

$$\begin{cases} p \in \text{CORE0oneNext} \\ p \notin \text{CORE0zeroNext} \\ p \notin \text{CORE0coreNext} \\ p \notin \text{CORE1Next} \\ p \notin \text{CORE2Next} \\ p \notin \text{ZEROAllNext} \\ p \notin \text{ONEAllNext} \end{cases}$$

Condition 3:

$$\begin{cases} p \in \text{CORE1zeroNext} \\ p \notin \text{CORE1oneNext} \\ p \notin \text{CORE1coreNext} \\ p \notin \text{CORE0Next} \\ p \notin \text{CORE2Next} \\ p \notin \text{ZEROAllNext} \\ p \notin \text{ONEAllNext} \end{cases}$$

Condition 4:

$$\begin{cases} p \in \text{CORE1oneNext} \\ p \notin \text{CORE1zeroNext} \\ p \notin \text{CORE1coreNext} \\ p \notin \text{CORE0Next} \\ p \notin \text{CORE2Next} \\ p \notin \text{ZEROAllNext} \\ p \notin \text{ONEAllNext} \end{cases}$$

Condition 5:

$$\begin{cases} p \in \text{CORE2zeroNext} \\ p \notin \text{CORE2oneNext} \\ p \notin \text{CORE2coreNext} \\ p \notin \text{CORE0Next} \\ p \notin \text{CORE1Next} \\ p \notin \text{ZEROAllNext} \\ p \notin \text{ONEAllNext} \end{cases}$$

Condition 6:

$$\begin{cases} p \in \text{CORE2oneNext} \\ p \notin \text{CORE2zeroNext} \\ p \notin \text{CORE2coreNext} \\ p \notin \text{CORE0Next} \\ p \notin \text{CORE1Next} \\ p \notin \text{ZEROAllNext} \\ p \notin \text{ONEAllNext} \end{cases}$$

It can be easily observed that any of the 6 conditions is used to ensure that only one bit condition on s^1 will determine whether v_0 will be next to v_1, which is irrelevant with the conditions on s^0 since we consider candidates from the propagation of core bits. Similar to previous three types of conditional cube tester, we have to further verify whether the quadratic term will appear in z^3 if the specified condition does not hold. Only then can we finally determine the position for p.

With such a searching method, we can recover extra 43 secret bits of s^1. The parameters for TYPE-IV conditional cube tester are given in Table 6. We give an explanation here. Take the first choice for instance. The cube variable v_0 is set at $s^0[1]$ and v_1 is set at $s^2[190]$. Note that flipping $s^0[1]$ will have no influence on the value of $s^1[213]$. If the condition that $s^1[213] = 1$ does not hold, then the cube sum of z^3 is nonzero with probability 1. If it holds, the cube sum is zero with probability 1. Thus, we can recover $s^1[213]$ as follows based on the cube sum of z^3.

$$\sum z^3 = 0 \Rightarrow s^1[213] = 1.$$
$$\sum z^3 \neq 0 \Rightarrow s^1[213] = 0.$$

Experimental Verification. We have implemented the four types of conditional cube tester and can successfully recover the $24 + 2 + 27 = 53$ secret state bits of s^0 and 43 secret bits of s^1. In each test, we will randomly generate 100000 examples of (s^0, s^1). Our experiments show that the 96 secret bits can always be correctly recovered for the 100000 random examples. Observe that by continuously performing the four types of conditional cube tester, i.e. continuously treating (s^i, s^{i+1}) as secret states, we can always recover 53 secret state bits of s^i and 43 secret bits of s^{i+1}. In other words, we can recover in total $53 + 43 = 96$ secret bits of s^i ($i \geq 1$).

Table 6: Parameters for TYPE-IV conditional cube tester

Position of v_0	1	1	2	4	11	15	15	17	17	22	35
Position of v_1	190	211	136	70	17	165	15	190	111	211	95
Position of condition	213	236	106	85	194	195	193	238	45	217	109
Value of condition	1	0	1	1	0	0	1	0	0	0	1
Position of v_0	35	64	64	70	95	111	111	128	128	136	140
Position of v_1	184	137	70	197	165	165	15	249	190	35	249
Position of condition	173	92	90	49	178	203	201	183	245	160	182
Value of condition	1	0	1	0	1	0	1	0	1	1	1
Position of v_0	165	169	169	184	184	184	189	190	190	197	197
Position of v_1	176	189	234	95	30	184	134	4	70	234	70
Position of condition	77	229	227	102	100	166	79	38	59	251	58
Value of condition	1	0	1	0	1	0	1	0	0	1	1
Position of v_0	213	213	223	225	225	225	234	234	249	249	–
Position of v_1	70	225	197	70	225	184	11	189	70	11	–
Position of condition	83	147	41	82	146	169	149	234	86	148	–
Value of condition	0	0	0	1	1	0	0	0	0	1	–

3.3 Recovering Full State

Based on the above method, 96 bits of the secret state s^1 can be recovered. Note that we can also extract 32-bit information $z^1 = extract(s^1)$, where

$$z^1[j] = s^1[\text{order0}[j]] \oplus s^1[\text{order1}[j]], \ (0 \le j \le 31).$$

Since some bits $s^1[\text{order0}[j]]$ and $s^1[\text{order1}[j]]$ $(0 \le j \le 31)$ are known, we can recover in total 111 bits of s^1. Moreover, we know extra $32 - 16 = 16$ linear equations of the secret state s^1. The recovered 111 secret bits are listed in Table 7, as marked in red. The time complexity to recover the 111 secret bits is $24\times2^2+2\times2^3+27\times2^2+43\times2^2 = 392$ times of encryption.

Now, we describe how to recover the full state. Set the nonce N and the associated data A as constants. Randomly choose a message longer than 128 bits denoted by M. The procedure to recover some secret state bits is described as below.

1. Send an encryption query (N, A, M) and obtain (C, T). Our goal is to recover the secret state $(MS_1^{in}, MS_2^{in}, MS_3^{in})$ in this query, as shown in Figure 2.
2. The first phase is to recover some bits of MS_1^{in} using TYPE-IV conditional cube tester. At this phase, we treat MS_0^{in}, MS_1^{in} and MS_2^{in} as s^0, s^1 and s^2 respectively. Based on the parameters of the parameters for TYPE-IV conditional cube tester in Table 6, we can recover 43 secret bits of MS_1^{in}.
3. The second phase is to recover some bits of MS_1^{in} and MS_2^{in}. At this phase, when asking an encryption query, the first message block has to be kept the same with that in the very first query. Then, we treat MS_1^{in}, MS_2^{in} and MS_3^{in} as s^0, s^1 and s^2 respectively. Based on the four types of conditional cube tester and their

corresponding parameters in Table 3, Table 4, Table 5 and Table 6, we can recover 53 extra secret bits of MS_1^{in} and 43 secret bits of MS_2^{in}.

4. The forth phase is to recover some bits of MS_2^{in} and MS_3^{in}. At this phase, when asking an encryption query, the first two message blocks have to be kept the same with those in the very first query. Then, we treat MS_2^{in}, MS_3^{in} and MS_4^{in} as s^0, s^1 and s^2 respectively. Using the four types of conditional cube tester, we can recover 53 extra secret bits of MS_2^{in} and 43 secret bits of MS_3^{in}.
5. The fifth phase is to recover some more bits of MS_3^{in}. At this phase, when asking an encryption query, the first three message blocks have to be kept the same with those in the very first query. Then, we treat MS_3^{in}, MS_4^{in} and MS_5^{in} as s^0, s^1 and s^2 respectively. Based on the first three types of conditional cube tester, 53 extra secret bits of MS_3^{in} can be recovered.

After the above procedure, we can know 111 secret bits and 16 linear equations of MS_1^{in}, MS_2^{in} and MS_3^{in}, respectively. Such a phase will require $3 \times (24 \times 2^2 + 2 \times 2^3 + 27 \times 2^2 + 43 \times 2^2) = 1176$ encryption queries. The recovered 111 bit positions are listed in Table 7, as marked in red.

Table 7: Information of known bits, secret bits and guessed bits, where the known bits are marked in red, the guessed bits are marked in blue, and the secret bits are marked in black.

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

Computing the Remaining Unknown Secret Bits of MS_1^{in}. Based on the above method, we can collect sufficient leaked bit information of MS_1^{in}, MS_2^{in} and MS_3^{in}. Now we explain how to use this leaked information to recover the full state. The main idea is to construct a quadratic boolean equation system which can be efficiently solved with the method of change of variables.

Since we have known 111 bits of MS_1^{in}, there will be $257 - 111 = 146$ unknown secret state bits, which can be treated as 146 unknown variables. Moreover, since the

degree of the one-round permutation is only 2, we can know that the 111 recovered bits of MS_2^{in} are quadratic in the 146 unknown variables. Moreover, since 111 bits of MS_2^{in} are known, we can know that some of the 111 bits of MS_3^{in} will be linear in MS_2^{in} and also quadratic in the 146 unknown variables. The reason is that $s_\chi^i[k-1]$ and $s_\chi^i[k-2]$ will be linear in s^i if $s^i[k]$ is known. We write the expression of the known bits of s^{i+1} quadratic in the 146 unknown variables when the 111 bits of s^i are recovered for a better understanding.

$$s^{i+1}[0] = s_\chi^i[0] \oplus s_\chi^i[3] \oplus s_\chi^i[8] \oplus 1. \quad (1)$$
$$s^{i+1}[1] = s_\chi^i[12] \oplus s_\chi^i[15] \oplus s_\chi^i[20]. \quad (2)$$
$$s^{i+1}[3] = s_\chi^i[36] \oplus s_\chi^i[39] \oplus s_\chi^i[44]. \quad (3)$$
$$s^{i+1}[14] = s_\chi^i[168] \oplus s_\chi^i[171] \oplus s_\chi^i[176]. \quad (4)$$
$$s^{i+1}[16] = s_\chi^i[192] \oplus s_\chi^i[195] \oplus s_\chi^i[200]. \quad (5)$$
$$s^{i+1}[23] = s_\chi^i[19] \oplus s_\chi^i[22] \oplus s_\chi^i[27]. \quad (6)$$
$$s^{i+1}[29] = s_\chi^i[91] \oplus s_\chi^i[94] \oplus s_\chi^i[99]. \quad (7)$$
$$s^{i+1}[41] = s_\chi^i[235] \oplus s_\chi^i[238] \oplus s_\chi^i[243]. \quad (8)$$
$$s^{i+1}[59] = s_\chi^i[194] \oplus s_\chi^i[197] \oplus s_\chi^i[202]. \quad (9)$$
$$s^{i+1}[65] = s_\chi^i[9] \oplus s_\chi^i[12] \oplus s_\chi^i[17]. \quad (10)$$
$$s^{i+1}[71] = s_\chi^i[81] \oplus s_\chi^i[84] \oplus s_\chi^i[89]. \quad (11)$$
$$s^{i+1}[82] = s_\chi^i[213] \oplus s_\chi^i[216] \oplus s_\chi^i[221]. \quad (12)$$
$$s^{i+1}[83] = s_\chi^i[225] \oplus s_\chi^i[228] \oplus s_\chi^i[233]. \quad (13)$$
$$s^{i+1}[100] = s_\chi^i[172] \oplus s_\chi^i[175] \oplus s_\chi^i[180]. \quad (14)$$
$$s^{i+1}[135] = s_\chi^i[78] \oplus s_\chi^i[81] \oplus s_\chi^i[86]. \quad (15)$$
$$s^{i+1}[136] = s_\chi^i[90] \oplus s_\chi^i[93] \oplus s_\chi^i[98]. \quad (16)$$
$$s^{i+1}[149] = s_\chi^i[246] \oplus s_\chi^i[249] \oplus s_\chi^i[254]. \quad (17)$$
$$s^{i+1}[164] = s_\chi^i[169] \oplus s_\chi^i[172] \oplus s_\chi^i[177]. \quad (18)$$
$$s^{i+1}[166] = s_\chi^i[193] \oplus s_\chi^i[196] \oplus s_\chi^i[201]. \quad (19)$$
$$s^{i+1}[170] = s_\chi^i[241] \oplus s_\chi^i[244] \oplus s_\chi^i[249]. \quad (20)$$
$$s^{i+1}[178] = s_\chi^i[80] \oplus s_\chi^i[83] \oplus s_\chi^i[88]. \quad (21)$$
$$s^{i+1}[182] = s_\chi^i[128] \oplus s_\chi^i[131] \oplus s_\chi^i[136]. \quad (22)$$
$$s^{i+1}[185] = s_\chi^i[164] \oplus s_\chi^i[167] \oplus s_\chi^i[172]. \quad (23)$$
$$s^{i+1}[191] = s_\chi^i[236] \oplus s_\chi^i[239] \oplus s_\chi^i[244]. \quad (24)$$
$$s^{i+1}[195] = s_\chi^i[27] \oplus s_\chi^i[30] \oplus s_\chi^i[35]. \quad (25)$$
$$s^{i+1}[196] = s_\chi^i[39] \oplus s_\chi^i[42] \oplus s_\chi^i[47]. \quad (26)$$

$$s^{i+1}[212] = s^i_\chi[231] \oplus s^i_\chi[234] \oplus s^i_\chi[239]. \quad (27)$$
$$s^{i+1}[217] = s^i_\chi[34] \oplus s^i_\chi[37] \oplus s^i_\chi[42]. \quad (28)$$
$$s^{i+1}[234] = s^i_\chi[238] \oplus s^i_\chi[241] \oplus s^i_\chi[246]. \quad (29)$$
$$s^{i+1}[235] = s^i_\chi[250] \oplus s^i_\chi[253] \oplus s^i_\chi[1]. \quad (30)$$
$$s^{i+1}[238] = s^i_\chi[29] \oplus s^i_\chi[32] \oplus s^i_\chi[37]. \quad (31)$$
$$s^{i+1}[242] = s^i_\chi[77] \oplus s^i_\chi[80] \oplus s^i_\chi[85]. \quad (32)$$
$$s^{i+1}[250] = s^i_\chi[173] \oplus s^i_\chi[176] \oplus s^i_\chi[181]. \quad (33)$$
$$s^{i+1}[255] = s^i_\chi[233] \oplus s^i_\chi[236] \oplus s^i_\chi[241]. \quad (34)$$
$$s^{i+1}[234] = s^i_\chi[238] \oplus s^i_\chi[241] \oplus s^i_\chi[246]. \quad (35)$$
$$s^{i+1}[165] = s^i_\chi[181] \oplus s^i_\chi[184] \oplus s^i_\chi[189]. \quad (36)$$
$$s^{i+1}[44] = s^i_\chi[14] \oplus s^i_\chi[17] \oplus s^i_\chi[22]. \quad (37)$$
$$s^{i+1}[129] \oplus s^{i+1}[193] = s^i_\chi[9] \oplus s^i_\chi[14] \oplus s^i_\chi[3] \oplus s^i_\chi[11]. \quad (38)$$
$$s^{i+1}[58] \oplus s^{i+1}[79] = s^i_\chi[182] \oplus s^i_\chi[190] \oplus s^i_\chi[177] \oplus s^i_\chi[180]. \quad (39)$$

Moreover, we write the expressions for extra two output bits.

$$z^{i+1}[7] = s^i_\chi[238] \oplus s^i_\chi[241] \oplus s^i_\chi[246] \oplus s^i_\chi[19] \oplus s^i_\chi[22] \oplus s^i_\chi[27]. \quad (40)$$
$$z^{i+1}[17] = s^i_\chi[132] \oplus s^i_\chi[135] \oplus s^i_\chi[140] \oplus s^i_\chi[125] \oplus s^i_\chi[128] \oplus s^i_\chi[133]. \quad (41)$$

For the above 41 equations, the terms on the right side will be quadratic in the 146 unknown variables, which can be easily verified according to Table 7.

Now we describe how practical it is to recover the remaining 146 unknown secret bits of MS^{in}_1. We guess 16 secret bits among the 146 unknown variables as follows, which are marked in blue in Table 7.

$$7, 9, 25, 27, 51, 53, 55, 57, 73,$$
$$75, 114, 116, 118, 123, 125, 151$$

In this way, after one round permutation, there are at most 54 possible quadratic terms formed by the remaining $146 - 16 = 130$ unknown variables. By replacing the 54 possible quadratic terms with 54 new variables, we can view the 130 unknown variables as $54 + 130 = 184$ variables. Note that we can know 16 extra linear equations of MS^{in}_1, 111 bits of MS^{in}_2, 16 extra linear equations of MS^{in}_2 and the above 41 quadratic equations in terms of the (not-guessed) 146 unknown variables. Thus, we can in total construct $16 + 111 + 16 + 41 = 184$ linear equations in terms of the new 184 variables. As a result, each guessed value of 16 secret state bits will suggest only one solution for the full state. We can check each solution by computing the corresponding ciphertext and tag obtained with the full state and compare it with the pre-obtained value. Only the correct value of the full state will remain. Thus, the time complexity to recover the full state is upper bounded by 2^{16}.

Remark. We note that it is possible to extract more equations based on the known bits of MS_3^{in}. Since the time complexity is very small and practical, we stop giving a further explanation.

Recovering the Secret Key. After the full secret state is recovered, we can compute backward until the state after K_3 is absorbed, denoted by KS_3^{ot}. In other words, we can know $257 - 32 = 225$ bits of KS_3^{ot}. Then, we can guess the 32-bit K_0 and 3 bits of K_1 that are injected at bit positions $(2, 136, 189)$. In this way, we can know that the state after K_2 is absorbed is linear in the remaining 29 secret bits of K_1 and the 32-bit K_2, thus making the 225 known bits of KS_3^{ot} quadratic in these $29 + 32 = 61$ variables. By computing the propagation of the 29 bits of K_1 for one-round permutation, we can easily count the quadratic terms formed by 61 secret variables and find that there are at most $123 + 3 = 126$ possible quadratic terms. Thus, by replacing these 126 quadratic terms with 126 new variables, we can know that the 225 known bits of KS_3^{ot} will be linear in the $126 + 61 = 187$ variables. In other words, we can construct an equation system of size 225 in terms of 187 variables. Only the right guess for the $32+3 = 35$ key bits will make this equation system have a solution. After the solution for (K_0, K_1, K_2) is obtained, combining with the recovered full state, we can compute the 32-bit K_3 and recover the full key. Hence, after the full state is recovered, the time complexity to recover the secret key is 2^{35}.

4 Distinguishing Attack on 4-Blank-Round Subterranean-SAE

Similar to the full-state recovery attack, we consider an equivalent presentation of the state transform as depicted in Figure 3. Suppose we are able to control 32 bits of s^0 and s^1. Moreover, only z^i $(i \geq 7)$ can be collected by the attacker. Now, we show how to construct a cube tester by setting cube variables at s^0 and s^1.

According to the array order0 in Table 2, the 32 controllable bit positions in s^0 and s^1 are as follows:

$$1, 2, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 136, 137,$$
$$140, 65, 169, 176, 184, 189, 190, 197, 211, 213, 223, 225, 234, 241, 249.$$

Therefore, by properly setting 29 cube variables in s^1, s^2 will be linear in the 29 cube variables. There are in total $2^3 = 8$ possible ways to choose these 29 cube variables. For example, setting $s^1[1]$, $s^1[136]$ and $s^1[189]$ to constants and other controllable bits to cube variables, then s^2 will be linear in these 29 cube variables. Denote the 29 cube variables set in s^1 by v_i^1 $(0 \leq i \leq 28)$. Next, we expect that there will be 4 cube variables in s^0 denoted by v_i^0 $(0 \leq j \leq 3)$ which satisfy the following constraints:

Constraint 1: v_j^0 $(0 \leq j \leq 3)$ are not next to each other in s^0, i.e. they do not multiply with each other after one-round permutation.

Constraint 2: After one more round permutation for v_j^0 $(0 \leq j \leq 3)$, none of them are next to any of v_i^1 $(0 \leq i \leq 28)$. Moreover, v_j^0 $(0 \leq j \leq 3)$ are still not next to each other.

With the tracing algorithm in section 3, we can easily obtain the $3 \times 3 = 9$ influenced bit positions in s^1 for each possible cube variable set in s^0. Then based on whether the 9 influenced bit positions are next to any of the 32 controllable bit positions in s^1, we can directly determine a candidate for the cube variable set in s^0. With such an idea to determine candidates, we obtain 5 valid bit positions in s^0 as follows:

$$30, 137, 189, 190, 223.$$

In other words, if we set five cube variables in $s^0[i]$ ($i \in \{3, 137, 189, 190, 223\}$), after one round permutation, none of them will propagate to the positions which are next to the 32 controllable bit positions in s^1 and they will not be next to each other either. Of course, the bit positions 189 and 190 cannot be chosen simultaneously. Moreover, we also observe that if $s^1[136]$ is set to a constant, then a cube variable set in $s^0[111]$ will not propagate to the positions which are next the remaining 31 controllable bit positions in s^1 nor next to the above five cube variables. Note that our goal is to select only 4 positions in s^0 for v_i^0 ($0 \leq j \leq 3$). Thus, we can easily find a solution, as displayed in Table 8. Our experiments have shown that the analysis is correct.

Table 8: Cube variables for cube tester

Bit positions in s^0	30, 111, 137, 223
Bit positions in s^1	2, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 190, 197, 211, 213, 223, 225, 234, 241, 249

With the 33 cube variables in Table 8, s^2 will be linear in them. Since the degree of one round permutation is 2, the cube sum of z^7 will always be zero. Now we describe how to construct a distinguisher for Subterranean-SAE when the number of blank rounds is reduced to 4.

Step 1: Set associated data empty and the first message block M_0 as a constant.
Step 2: Treat N_2, N_3 as s^0, s^1 respectively. When the number of blank rounds is reduced to 4, we can treat the ciphertext block C_0 as z^7, as shown in Figure 8. According to Table 8, send= 2^{33} encryption queries (N, A, M) with N taking all possible 2^{33} values and compute the sum of $C[0]$. The sum will always be zero.

Complexity Evaluation. Since we need to send 2^{33} encryption queries (N, A, M) with different N, the data and time complexity are both 2^{33}.

5 Key-recovery Attack on 4-Blank-Round Subterranean-SAE

When the number of blank rounds is reduced to 4, a key-recovery attack will be feasible. The attack procedure can be divided into two steps on the whole.

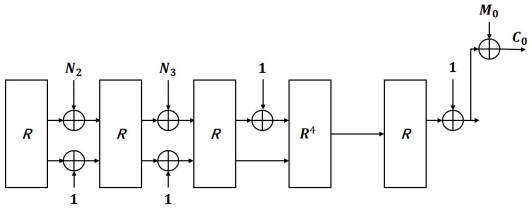

Fig. 8: Cube tester for 4-blank-round Subterranean-SAE

Step 1: With a similar idea of the full-state recovery attack, recover y secret bits of the state after N_1 is absorbed.

Step 2: Guess $(128 - x)$ bits of the secret key and let the remaining x secret key bits as variables. Then with the y recovered state bits, construct a quadratic boolean equation system in terms of the x variables. There will be $\frac{x(x-1)}{2}$ quadratic terms and we replace them with $\frac{x(x-1)}{2}$ new variables. In this way, we can obtain y linear equations in terms of $x + \frac{x(x-1)}{2}$ variables. If $y \geq x + \frac{x(x-1)}{2}$, the x secret key bits can be computed by solving this linear equation system.

To make this part clear, similar to the distinguishing attack, we first consider an equivalent representation of the state transform. In our distinguishing attack, the cube sum is always zero, which cannot help recover extra secret information. Thus, we hope the cube sum will depend on the value of one secret state bit as in the full-state recovery attack. Then, according to the cube sum, we can directly obtain one secret state bit.

While the cube variables are set at s^0 and s^1 and the attacker can only get z^i $(i \geq 7)$ in the distinguishing attack, we will set cube variables at s^i $(0 \leq i \leq 2)$ and suppose the attacker can only get z^i $(i \geq 8)$ in the key-recovery attack. The main idea can be briefly described as follows:

1. Set 32 cube variables in s^2, denoted by $v_j^2 = s^2[\text{order0}[j]]$ $(0 \leq j \leq 31)$.
2. Set n cube variables in s^1, denoted by $v_j^1 = s^1[\text{order0}[r]]$ where $0 \leq j < n$ and $r \in \{k | 0 \leq k \leq 31\}$.
3. Set $33 - n$ cube variables in s^0, denoted by $v_j^0 = s^0[\text{order0}[r]]$ where $0 \leq j < 33 - n$ and $r \in \{k | 0 \leq k \leq 31\}$.

Suppose $f(s^0[x])$ represents either $s^0[x]$ or $s^0[x] \oplus 1$. There will be some constraints on v^0 and v^1 as follows:

Constraint 1: v^0 are not next to each other in s^0, i.e. they will not multiply with each other after one-round permutation.

Constraint 2: v^1 are not next to each other in s^1, i.e. they will not multiply with each other after one-round permutation.

Constraint 3: If the specified bit condition $f(s^0[x]) = 0$ holds, after one-round permutation for v^0, none of v^0 will be next to any of v^1.

Constraint 4: If the specified bit condition $f(s^0[x]) = 0$ does not hold, after one-round permutation for v^0, v^0 will be next to at least one of v^1.

With the above constraints, we can know that s^2 will be linear in (v^0, v^1) if $f(s^0[x]) = 0$ holds. Observe that the algebraic degree of z^8 is at most $2^6 = 64$ in terms of s^2 and there are extra 32 cube variables in s^2. Hence, if $f(s^0[x]) = 0$ holds, the degree-65 term $v^0v^1v^2$ will not appear in the expression of z^8 and the cube sum of z^8 will be zero.

However, when the condition does not hold, s^2 will contain a quadratic term. Then, the degree-65 term $v^0v^1v^2$ is expected to appear in the expression of z^8 due to the sufficient diffusion for the cube variables. For this case, the cube sum of z^8 cannot be predicted.

Consequently, according to the cube sum, we can directly recover the one secret bit $s^0[x]$ as the full-state recovery attack. Combining the methods to select cube variables for full-state recovery attack and distinguishing attack, we can find 22 valid choices for (v^0, v^1) and therefore recover 22 secret bits of s^0, as listed in Table 9 and Table 10 in Appendix A. For a better understanding of the two tables, we take the first choice in Table 9 for instance and give an explanation.

For the first choice in Table 9 to recover the secret state $s^0[2]$, the cube variables v^0 are set at 6 bit positions of s^0 and v^1 are set at 27 bit positions of s^1. Specifically,

$$
\begin{aligned}
&v^0_0 = s^0[1], v^0_1 = s^0[30], v^0_2 = s^0[111], v^0_3 = s^0[137], v^0_4 = s^0[189], v^0_5 = s^0[233],\\
&v^1_0 = s^1[1], v^1_1 = s^1[4], v^1_2 = s^1[11], v^1_3 = s^1[15], v^1_4 = s^1[17],\\
&v^1_5 = s^1[22], v^1_6 = s^1[30], v^1_7 = s^1[35], v^1_8 = s^1[64], v^1_9 = s^1[70],\\
&v^1_{10} = s^1[95], v^1_{11} = s^1[111], v^1_{12} = s^1[128], v^1_{13} = s^1[134], v^1_{14} = s^1[137],\\
&v^1_{15} = s^1[140], v^1_{16} = s^1[165], v^1_{17} = s^1[169], v^1_{18} = s^1[176], v^1_{19} = s^1[184],\\
&v^1_{20} = s^1[189], v^1_{21} = s^1[197], v^1_{22} = s^1[211], v^1_{23} = s^1[223], v^1_{24} = s^1[225],\\
&v^1_{25} = s^1[241], v^1_{26} = s^1[249].
\end{aligned}
$$

Once the condition $s^0[2] = 0$ holds, the cube sum of z^8 is zero. However, when $s^0[2] \neq 0$, three bits of s^2 will always contain a quadratic term $v^0_0v^1_0$. Moreover, similar to the full-state recovery attack, we have verified that there will always be a cubic term in a certain bit of s^3. Since there are 65 cube variables and sufficient number of rounds to diffuse v^0, v^1 and v^2, we expect there will be a degree-65 term in z^8. Therefore, based on the cube sum of z^8, we directly recover the secret state bit $s^0[2]$ as follows:

$$
\begin{aligned}
\sum z^8 \neq 0 &\Rightarrow s^0[2] = 1,\\
\sum z^8 = 0 &\Rightarrow s^0[2] = 0.
\end{aligned}
$$

Now, we describe how to use the above method to recover the secret state after N_1 is absorbed. Set the associated data A as empty and the first message block M_0 as a zero constant. Denote the state after N_i is absorbed as NS^{in}_i, as depicted in Figure 9. The attack procedure can be described as follows:

Step 1: Send an encryption query (N, A, M) and obtain (C, T).

Step 2: Keep M_0 and N_0 as constant. Treat NS^{in}_1, NS^{in}_2 and NS^{in}_3 as s^0, s^1 and s^2 respectively. For each choice of the 65 cube variables in Table 9 and Table 10, send 2^{65} encryption queries (N, A, M) with N taking all possible 2^{65} values and compute the sum of C_0. If the sum is zero, the corresponding condition will

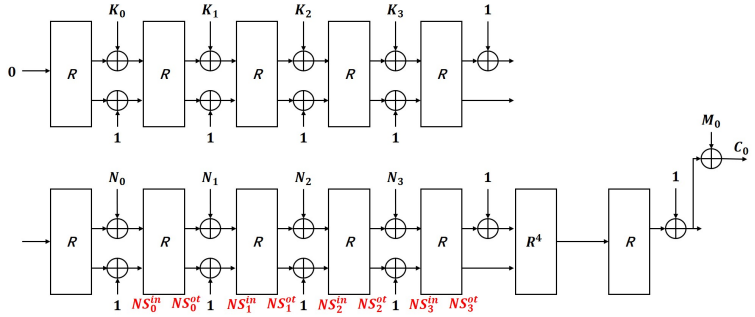

Fig. 9: Key recovery attack

hold. If it is not zero, the condition will not hold. Whatever the sum is, we can recover one secret bit of NS_0^{ot}. The time and data complexity to recover the 22 secret bits of NS_0^{ot} are both $22 \times 2^{65} = 2^{69.5}$.

After recovering the 22 secret bits of NS_0^{ot}, we will start to construct 22 equations. Suppose K_0, K_1 and K_2 are fixed, we then use a trivial MILP-based method to find the maximum number of variables in K_3 which are still linear after two-round permutation and the Gurobi solver returns 9. The 9 positions are listed below:

$$11, 35, 70, 95, 140, 165, 190, 213, 241.$$

In other words, if we fix the remaining $32 - 9 = 23$ bits of K_3 as constants, NS_0^{in} will be linear in these 9 secret bits. Since NS_0^{ot} is quadratic in NS_0^{in}, we therefore cannot construct a linear equation system. Guessing 3 more bits among the 9 secret bits will reduce the number of variables to 6. Therefore, there will be $6\times(6-1)/2 = 15$ quadratic terms. By replacing the 15 quadratic terms with 15 new variables, we can now know that NS_0^{ot} is linear in the $6 + 15 = 21$ variables. Since 22 bits of NS_0^{ot} have been recovered, we can construct 22 linear equations in terms of 21 variables. It is expected there is only one solution for each guess of K_i ($0 \le i \le 3$). For each solution, we compute the tag T' and the corresponding ciphertext C'. Only when $T = T'$ and $C' = C$ will imply that the recovered key is correct.

Complexity Evaluation. The key-recovery attack is divided into two steps. The first step is to recover 22 secret state bits. The time complexity and data complexity at this step is $22 \times 2^{65} \approx 2^{69.5}$. After the 22 secret bits are recovered, we will start the second step. At this step, we will guess 122 bits of the secret key and let the remaining 6 key bits keep as variables. For each guess of the 122 secret key bits, we can construct a linear equation system of size 22 to compute the 6 unknown key bits with Gauss elimination. The time to solve this equation system is negligible. Moreover, we expect there is only one solution for this linear equation system. After the 6 unknown key bits are computed, the key is known and we can compute the ciphertext and tag computed based on this key and compare it with the pre-obtained ciphertext and tag. The probability that they match with each other is lower than 2^{-128}. Therefore, only the correct key will remain

and the time complexity of the second step is 2^{122}. In total, the time complexity and date complexity of key-recovery attack are 2^{112} and $2^{69.5}$, respectively.

Remark. As can be observed in our full-state recovery and key-recovery attack, the problem is finally reduced to solving a quadratic boolean equation system. One may claim that this can be solved with existing state-of-the-art solvers. However, as pointed out by many papers, the performance to solve the quadratic (or higher degree) boolean equation system is instable. Hence, the time complexity of our method by re-linearizing or change of variables to convert the quadratic boolean equation system into a linear equation system can be viewed as an upper bound.

6 Conclusion

The designers of Subterranean 2.0 expect that it may require a non-trivial effort to mount a full-state recovery attack for Subterranean-SAE in the nonce-misuse scenario. Following this expectation, we make the first effort to achieve it with practical time complexity 2^{16}. In addition, the same nonce is only required to be reused for 1177 times. Moreover, to investigate the security provided by the number of blank rounds, we consider the reduced variant of Subterranean-SAE by reducing the number of blank rounds to 4 from 8. For such a variant, a distinguishing attack can be achieved with time and data complexity 2^{33}. The key-recovery attack with time complexity 2^{122} and data complexity $2^{69.5}$ is also faster than brute force for this variant. We hope our cryptanalysis can advance the understanding of Subterranean-SAE.

Acknowledgement We thank Joan Daemen for a discussion on the results in this paper and providing insightful comments.

References

1. Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube testers and key recovery attacks on reduced-round MD6 and Trivium. In *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium, February 22-25, 2009, Revised Selected Papers*, pages 1–22, 2009.
2. Joan Daemen, Pedro Maat Costa Massolino, and Yann Rotella. The Subterranean 2.0 cipher suite, 2019. `https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates`.
3. Itai Dinur and Adi Shamir. Cube attacks on tweakable black box polynomials. In *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*, pages 278–299, 2009.
4. Senyang Huang, Xiaoyun Wang, Guangwu Xu, Meiqin Wang, and Jingyuan Zhao. Conditional cube attack on reduced-round Keccak sponge function. In *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*, pages 259–288, 2017.

5. Mustafa Khairallah. Forgery attack on SNEIKEN. Cryptology ePrint Archive, Report 2019/408, 2019. https://eprint.iacr.org/2019/408.
6. Zheng Li, Wenquan Bi, Xiaoyang Dong, and Xiaoyun Wang. Improved conditional cube attacks on Keccak keyed modes with MILP method. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 99–127, 2017.
7. Zheng Li, Xiaoyang Dong, Wenquan Bi, Keting Jia, Xiaoyun Wang, and Willi Meier. New conditional cube attack on Keccak keyed modes. *IACR Trans. Symmetric Cryptol.*, 2019(2):94–124, 2019.
8. Fukang Liu, Zhenfu Cao, and Gaoli Wang. Finding ordinary cube variables for Keccak-MAC with greedy algorithm. Cryptology ePrint Archive, Report 2018/799, 2018. To appear at IWSEC 2019. https://eprint.iacr.org/2018/799.
9. Fukang Liu and Takanori Isobe. Iterative differential characteristic of trifle-bc. Cryptology ePrint Archive, Report 2019/727, 2019. To appear at SAC 2019. https://eprint.iacr.org/2019/727.
10. Léo Perrin. Probability 1 iterated differential in the SNEIK permutation. Cryptology ePrint Archive, Report 2019/374, 2019. https://eprint.iacr.org/2019/374.
11. Ling Song, Jian Guo, Danping Shi, and San Ling. New MILP modeling: Improved conditional cube attacks on Keccak-based constructions. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, pages 65–95, 2018.
12. Yosuke Todo, Takanori Isobe, Yonglin Hao, and Willi Meier. Cube attacks on non-blackbox polynomials based on division property. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, pages 250–279, 2017.
13. Yosuke Todo and Masakatu Morii. Bit-based division property and application to simon family. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, pages 357–377, 2016.
14. Qingju Wang, Yonglin Hao, Yosuke Todo, Chaoyun Li, Takanori Isobe, and Willi Meier. Improved division property based cube attacks exploiting algebraic properties of superpoly. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part I*, pages 275–305, 2018.

A Algorithm and Tables

We present the algorithm and some tables in this section.

Table 9: Cube variables for conditional cube tester

Bit positions in s^0	1, 30, 111, 137, 189, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 223, 225, 241, 249
condition	$s^0[2] = 0$
Bit positions in s^0	2, 30, 137, 189,
Bit positions in s^1	2, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 213 , 223, 225, 234 , 241, 249
condition	$s^0[3] = 0$
Bit positions in s^0	2, 30, 111, 137, 189, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 223, 225, 241, 249
condition	$s^0[1] = 1$
Bit positions in s^0	4, 30, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[5] = 0$
Bit positions in s^0	11, 30, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[10] = 1$
Bit positions in s^0	15, 137, 189, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[16] = 0$
Bit positions in s^0	22, 111, 137, 189, 223,
Bit positions in s^1	2, 4, 11, 15, 17, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[21] = 1$
Bit positions in s^0	64, 30, 111, 137, 189, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 128, 137, 140, 165, 169, 176, 184, 189, 197 , 211, 213, 223, 225, 234, 241, 249
condition	$s^0[65] = 0$
Bit positions in s^0	64, 30, 111, 137, 189, 223,
Bit positions in s^1	1, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[63] = 1$
Bit positions in s^0	70, 30, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176 , 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[69] = 1$
Bit positions in s^0	95, 30, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 136, 140, 165, 169, 176, 184, 189, 197, 211, 213, 223 , 225, 234, 241, 249
condition	$s^0[96] = 0$

Table 10: Cube variables for conditional cube tester

Bit positions in s^0	111, 30, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 136, 140, 165, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[112] = 0$
Bit positions in s^0	134, 30, 111, 189, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 165, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[133] = 1$
Bit positions in s^0	136, 30, 189, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[135] = 1$
Bit positions in s^0	165, 30, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184 , 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[166] = 0$
Bit positions in s^0	184, 30, 137, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[185] = 0$
Bit positions in s^0	197, 30, 111, 137, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 169, 176, 184, 189, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[196] = 1$
Bit positions in s^0	211, 30, 137, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 136, 140, 165, 169, 176, 184, 189, 197, 211 , 213, 223, 225, 234, 241, 249
condition	$s^0[212] = 0$
Bit positions in s^0	213, 30, 137, 223,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 136, 140, 165, 169, 176, 184, 190 , 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[214] = 0$
Bit positions in s^0	225, 30, 111, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 176, 184, 189 , 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[226] = 0$
Bit positions in s^0	241, 30, 111, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 176, 184, 190 , 197, 211 , 213 , 223, 225, 234, 241, 249
condition	$s^0[240] = 1$
Bit positions in s^0	249, 30, 111, 137, 189,
Bit positions in s^1	1, 4, 11, 15, 17, 22, 30, 35, 64, 70, 95, 111, 128, 134, 137, 140, 165, 176, 184, 190, 197, 211, 213, 223, 225, 234, 241, 249
condition	$s^0[248] = 1$

Algorithm 3 Determine candidates of cube variables for TYPE-I conditional cube tester

```
vector<> candidate
int V1Pos, conditionValue
int zero[], one[], core[]
int zeroSize=0, oneSize=0, coreSize=0
for i from 0 to 2 do
    if CORE[i]-1 ∈ {e|e = order0[j], 0 ≤ j ≤ 31} then
        core[coreSize]=CORE[i]-1
        coreSize++;
    end if
    if CORE[i]+1 ∈ {e|e = order0[j], 0 ≤ j ≤ 31} then
        core[coreSize]=CORE[i]+1;
        coreSize++;
    end if

    if ZERO[i]-1 ∈ {e|e = order0[j], 0 ≤ j ≤ 31} then
        zero[zeroSize]=ZERO[i]-1
        zeroSize++;
    end if
    if ZERO[i]+1 ∈ {e|e = order0[j], 0 ≤ j ≤ 31} then
        zero[zeroSize]=ZERO[i]+1;
        zeroSize++;
    end if

    if ONE[i]-1 ∈ {e|e = order0[j], 0 ≤ j ≤ 31} then
        one[oneSize]=ONE[i]-1
        oneSize++;
    end if
    if ONE[i]+1 ∈ {e|e = order0[j], 0 ≤ j ≤ 31} then
        one[oneSize]=ONE[i]+1;
        oneSize++;
    end if
end for

for i from 0 to zeroSize-1 do
    if zero[i] ∉ core and zero[i] ∉ one then
        v1Pos=zero[i]
        conditionValue=0
        candidate.pushback([v1Pos,conditionValue])
    end if
end for

for i from 0 to oneSize-1 do
    if one[i] ∉ core and one[i] ∉ zero then
        v1Pos=one[i]
        conditionValue=1
        candidate.pushback([v1Pos,conditionValue])
    end if
end for
return candidate
```