Conditional Cube Attack on Reduced-Round Keccak Sponge Function

Senyang Huang[1], Xiaoyun Wang[1,2,3⋆], Guangwu Xu[4], Meiqin Wang[2,3], Jingyuan Zhao[5]

[1] Institute for Advanced Study, Tsinghua University, Beijing 100084, China
[2] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Jinan 250100, China
[3] School of Mathematics, Shandong University, Jinan 250100, China
[4] Computer Science, University of Wisconsin-Milwaukee, Milwaukee, WI 53201, USA
[5] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
xiaoyunwang@mail.tsinghua.edu.cn

Abstract. The security analysis of Keccak, the winner of SHA-3, has attracted considerable interest. Recently, some attention has been paid to the analysis of keyed modes of Keccak sponge function. As a notable example, the most efficient key recovery attacks on Keccak-MAC and Keyak were reported at EUROCRYPT'15 where cube attacks and cube-attack-like cryptanalysis have been applied. In this paper, we develop a new type of cube distinguisher, the *conditional cube tester*, for Keccak sponge function. By imposing some bit conditions for certain cube variables, we are able to construct cube testers with smaller dimensions. Our conditional cube testers are used to analyse Keccak in keyed modes. For reduced-round Keccak-MAC and Keyak, our attacks greatly improve the best known attacks in key recovery in terms of the number of rounds or the complexity. Moreover, our new model can also be applied to keyless setting to distinguish Keccak sponge function from random permutation. We provide a searching algorithm to produce the most efficient conditional cube tester by modeling it as an MILP (mixed integer linear programming) problem. As a result, we improve the previous distinguishing attacks on Keccak sponge function significantly. Most of our attacks have been implemented and verified by desktop computers. Finally we remark that our attacks on the the reduced-round Keccak will not threat the security margin of Keccak sponge function.

Keywords: Keccak-MAC, Keyak, cube tester, conditional cube variable, ordinary cube variable

1 Introduction

The Keccak sponge function family, designed by Bertoni, Daemen, Peeters, and Giles in 2007 [1], was selected by the U.S. National Institute of Standards and Technology (NIST) in 2012 as the proposed SHA-3 cryptographic hash function.

⋆ Corresponding author

Due to its theoretical and practical importance, cryptanalysis of Keccak has attracted increasing attention. There has been extensive research recently, primarily on the keyless setting. For example, in keyless modes of reduced-round Keccak, many results have been obtained on collision attack [2], preimage attack [3] and second preimage attack [4]. Additionally, there are also some research focused on the distinguishers of Keccak internal permutation, in which the size of input is the full state. In [5], a distinguisher of full 24-round Keccak internal permutation was proposed which takes 2^{1579} Keccak calls. Using the rebound attack and efficient differential trails, Duc et al.[6] derived a distinguisher for 8-round Keccak internal permutation with the complexity 2^{491}. Jérémy et al.[7] provided an 8-round internal differential boomerang distinguisher on Keccak with practical complexity. It should be remarked that these results on Keccak internal permutation seem to be a little far from the security margin of Keccak sponge function, which do not lead any attacks to Keccak hash function. For distinguishing attacks on Keccak sponge function with the bitrate part as its input, some results have been given in [8], [9] and [10]. These distinguishers are one step closer to the security margin but some of these distinguisher are far from being practical.

By embedding a secret key in a message as an input, Keccak can be used in several settings. For example, Keccak sponge function can produce a pseudo-random binary string of arbitrary length, and hence can serve as a stream cipher. It is also a natural keyed hash function, namely, a message authentication code (MAC). Moreover, an authenticated encryption (AE) scheme based on Keccak was described in [11]. However, there is much less research reported for the keyed modes of the family of Keccak sponge functions. Besides the side channel attack for Keccak-MAC [12], the celebrated paper on key recovery attacks [10] seems to be the only one found in the literature for analysing keyed modes of Keccak. In [10], the authors set cube variables in the CP kernel to control the propagation of the mapping θ in the first round. More specifically, the cube dimension can be reduced by carefully selecting cube variables so that they are not multiplied with each other after the first round. The cube sums of output polynomials depend only on a portion of key bits. The dedicated cube-attack-like cryptanalysis uses this property to construct the first key recovery attack on reduced-round Keccak-MAC and Keyak. It is also noted that the cube attack and cube-attack-like are very efficient techniques in analysing Keccak-like cryptosystems in [13] and [14].

We observe that most of the attacks described in the previously published work deal with propagations of cube variables only after the first round. Thus, it is a natural and interesting question to ask whether and how we can control certain relations of cube variables after the second round of the Keccak sponge function to push this kind of attacks further. The purpose of this paper is to answer this question by proposing the technique of *conditional cube tester* and making the corresponding attacks more efficient. To the best of our knowledge, the results obtained in this paper are currently the best in terms of the number of rounds or the complexity.

1.1 Our Contributions

Conditional Cube Tester for Keccak Sponge Function. Our conditional cube tester model is inspired by the dynamic cube attack on Grain stream cipher [15]. The approach of dynamic cube attack in [15] is to set some bit conditions on the initial value (IV) so that the intermediate polynomials can be simplified and the degree of output polynomial can be reduced. However, this approach cannot be utilized directly in the setting of Keccak sponge function because its structure is very different from that of Grain stream cipher. Additionally, the number of intermediate polynomials related to the ones in the previous round is too large for Keccak, which makes the approach of dynamic cube attack infeasible. In this paper, we propose a new approach by imposing bit conditions on the input to control the propagation of cube variables caused by the nonlinear operation χ. This will be helpful in identifying the cube variables that are not multiplied with each other after the second round of Keccak sponge function. We provide several algorithms for searching the cube variables and imposing the corresponding bit conditions. These algorithms give a base to construct a conditional cube tester. In some cases the dimension of this cube tester is smaller than the cube testers in [10]. Our model is also influenced by the conditional differential cryptanalysis method developed in [16].

Improved Key Recovery Attack on Reduced-Round Keccak-MAC. We have obtained improved results for Keccak-MAC by applying the conditional cube tester. For 5-round Keccak-MAC-512, our key recovery attack makes 2^{24} Keccak calls. We are also able to recover full key bits for 6-round Keccak-MAC-384 with the complexity of 2^{40}. Furthermore, we prove that a 7-round Keccak-MAC-256 can be broken using 2^{72} Keccak calls. These results greatly improve the current best complexity bounds for key recovery attacks reported in [10]. Notice that in [10] the attacks were performed on 5-round Keccak-MAC-288 and 6-round and 7-round Keccak-MAC-128, with the time complexity of 2^{35}, 2^{66} and 2^{97} respectively. As it is easy to see that an attack on Keccak-MAC-n_1 can be used to break Keccak-MAC-n_2 without increasing its complexity as long as $n_1 \geq n_2$, we conclude that our attacks cover those in [10] with better efficiencies. It is remarked that our attacks on 5-round Keccak-MAC-512 and 6-round Keccak-MAC-384 are practical and have been verified by experiments. In Table 1, we list a comparison of the performance of our key recovery attacks and the existing ones. This table also shows that our attacks save the space complexity significantly.

Improved Key Recovery Attack on Reduced-Round Keyak. Keyak is an AE scheme based on Keccak sponge function [11]. In this paper we also use the technique of conditional cube tester to recover the key for reduced-round Keyak. In this situation, we assume that a message is of two blocks and the nonce could be reused. This means that our attacks on Keyak break the properties of authenticity and integrity because the specification of Keyak [11]

Rounds	Capacity	Time	Data	Memory	Reference
5	576	2^{35}	2^{35}	negligible	[10]
6	256	2^{66}	2^{64}	2^{32}	[10]
7	256	2^{97}	2^{64}	2^{32}	[10]
5	576/1024	2^{24}	2^{24}	negligible	Section 4
6	256/768	2^{40}	2^{40}	negligible	Section 4
7	256/512	2^{72}	2^{72}	negligible	Section 4

Table 1. Summary of key recovery attacks on Keccak-MAC

states that a nonce may not be variable when only authenticity and integrity are required. We perform our attacks on 7-round and 8-round Keyak with the time complexity of 2^{42} and 2^{74} respectively. Under the same assumption on the nonce, [10] proposed a key recovery attack on Keyak, which can work up to 7 rounds with the time complexity of 2^{76}. Table 2 compares our results with the existing attacks on Keyak, and shows a significant reduction of complexity by using our method. It is also interesting to note that the memory complexity in our attacks is negligible.

Rounds	Capacity	Time	Data	Memory	Reference
7	256	2^{76}	2^{75}	2^{43}	[10]
7	256	2^{42}	2^{42}	negligible	Section 5
8	256	2^{74}	2^{74}	negligible	Section 5

Table 2. Summary of key recovery attacks on Keyak

Improved Distinguishing Attack on Keccak Sponge Function. In addition to the cases of keyed modes of Keccak, we use the technique of conditional cube tester in keyless setting as well. To be more specific, we use this technique to carry out distinguishing attacks on Keccak sponge function. With the help of mixed integer linear programming (MILP), we can get a suitable combination of conditional cube variables automatically with good efficiency. As a result, practical distinguishing attacks have been achieved for Keccak sponge function up to seven rounds. There have been several distinguishing attacks on Keccak sponge function reported in the published papers. In [8], Naya-Plasencia et al. put forward a 4-round differential distinguisher over Keccak-256/224. A 6-round distinguisher over Keccak-224 was constructed in [9] by Das et al. Recently, a straightforward distinguisher on n-round Keccak sponge function was given in [10] which invokes $2^{2^{n-1}+1}$ Keccak calls for $n \leq 7$. Table 3 lists these existing distinguishing attacks on Keccak sponge function together with our attacks. It can be seen that our improvements over the previous attacks are quite significant.

The remainder of the paper is organized as follows. We introduce some preliminaries needed for the paper in Section 2, including Keccak sponge function,

Rounds	Capacity	Time	Data	Memory	Referance
4	448/512	2^{25}	2^{24}	negligible	[8]
6	448	2^{52}	2^{52}	negligible	[9]
6	448/512/576	2^{33}	2^{33}	negligible	[10]
7	448/512/576	2^{65}	2^{65}	negligible	[10]
5	448/512	2^{9}	2^{9}	negligible	Section 6
6	768/1024	2^{9}	2^{9}	negligible	Section 6
6	448/512/576	2^{17}	2^{17}	negligible	Section 6
7	768	2^{17}	2^{17}	negligible	Section 6
7	448	2^{33}	2^{33}	negligible	Section 6

Table 3. Summary of distinguishing attacks on Keccak sponge function

two keyed modes of Keccak, and the idea of cube tester. In Section 3, we will describe our new model, the conditional cube tester. Key recovery attacks for Keccak-MAC and Keyak based on our new model will be discussed in detail in Section 4 and Section 5. Section 6 is devoted to distinguishing Keccak sponge function from a random permutation using the conditional cube tester. Finally, we conclude the paper in Section 7.

2 Preliminaries

In the section, we will briefly introduce some necessary background for this paper. We will describe Keccak sponge function including two keyed modes, namely Keccak-MAC and the AE scheme Keyak. In the later part of the section, the idea of cube tester will be described.

2.1 Keccak Sponge Function

Description of Keccak Sponge Function. We shall just describe the Keccak sponge function in its default version. We refer the readers to [1] for the complete Keccak specification.

The (default) sponge function works on a 1600-bit state A, which is simply a three-dimensional array of bits, namely $A[5][5][64]$. The one-dimensional arrays $A[\ ][y][z], A[x][\ ][z]$ and $A[x][y][\ ]$ are called a column, a row and a lane respectively; the two-dimensional array $A[\ ][\ ][z]$ is called a slice(see Fig. 1). The coordinates are always considered modulo 5 for x and y and modulo 64 for z. Each 1600-bit string a is interpreted as a state A in the following manner: the $(64(5y + x) + z)$th bit of a becomes $A[x][y][z]$.

For each $n \in \{224, 256, 384, 512\}$, the sponge function Keccak-n corresponds to parameters r (bitrate) and $c = 2n$ (capacity) with $r + c = 1600$. Initially, all the 1600 bits are filled with 0s and the message will be split into r-bit blocks. There are two phases in the Keccak sponge 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 internal permutation which consists of 24

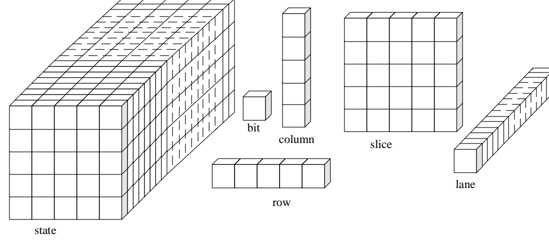


Fig. 1. Terminologies used in Keccak

rounds. After all the blocks are absorbed, the squeezing phase starts. In this phase, Keccak-n will return the first r bits as the output of the function with internal permutation iteratively until the n-bit digest is produced.

In the permutation, each round is computed by composing five operations θ, ρ, π, χ and ι as $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$. Given a round constant RC, the round function can be described by the following pseudo-code, where $r[x, y]$ is the offset of the internal permutation shown in Table 8 and for a lane L, $\text{rot}[L, n]$ means $L >>> n$.

```
R(A, RC)
{
    θ step
    for x in (0...4)
        C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4]
        D[x] = C[x-1] xor rot(C[x+1],1)
    for x in (0...4)
        for y in (0...4)
           A[x,y] = A[x,y] xor D[x]
    ρ step
    for x in (0...4)
        for y in (0...4)
            A[x,y] = rot[A[x,y],r[x,y]]
    π step
    for x in (0...4)
        for y in (0...4)
            B[y,2*x+3*y] = A[x,y]
    χ step
    for x in (0...4)
        for y in (0...4)
            A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y])
    ι step
    A[0,0] = A[0,0] xor RC
```

```
    return A
};
```

The purpose of θ is to diffuse the state. If a variable in every column of state has even parity, it will not diffuse to other columns: this is the *column parity kernel (CP kernel)* property. Thus diffusion of some input variables caused by θ can be controlled in the first round. This property has been widely used in cryptanalysis of Keccak. For example, the attacks in [10] use it to decrease the dimension of the cube. The operations ρ and π just change the position of bits. The first three linear operations θ, ρ and π will be called half a round. In the permutation, the only nonlinear operation is χ whose algebraic degree is 2. Therefore, after an n-round Keccak internal permutation, the algebraic degree of the output polynomial is at most 2^n. We will not consider ι since it has no impact on our attacks.

2.2 Keyed Modes of Keccak

MAC based on Keccak. As an example demonstrated in Fig. 2, one gets a MAC (or a tag) by concatenating a secret key with a message as the input to a hash function. This primitive to ensure data integrity and authentication of a message should satisfy the two following security requirements: no key recovery and resistance of MAC forgery.

Fig. 2 shows the construction of Keccak-MAC-n working on a single block. As described in 2.1, n is half of capacity length. In this paper we will use a single block message and assume that the key and tag are 128 bits long. So there are two significant lanes that consist of key bits. Block sizes may be different based on the variants we analyse.

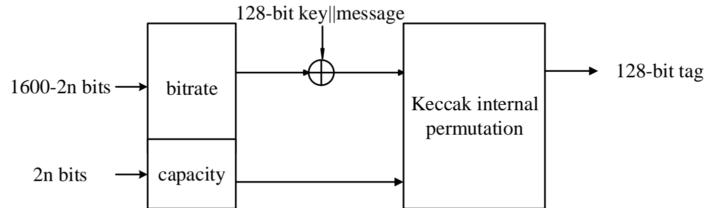


Fig. 2. Construction of Keccak-MAC-n

Authenticated Encryption Scheme based on Keccak An AE scheme is used to provide confidentiality, integrity and authenticity of data where decryption is combined with integrity verification. An authenticated encryption scheme

based on Keccak is the scheme Keyak [11] which is a third-round candidate algorithm submitted to CAESAR [17]. Fig.3 depicts the construction of Keyak on two-block message. Both key and nonce are 128 bits. The capacity is 256 bits long and the bitrate is 1344 bits long.

According to the specification of Keyak [11], when confidentiality of data is not required, a nonce can be reused. In this paper, we shall restrict our discussion to the two-block Keyak.

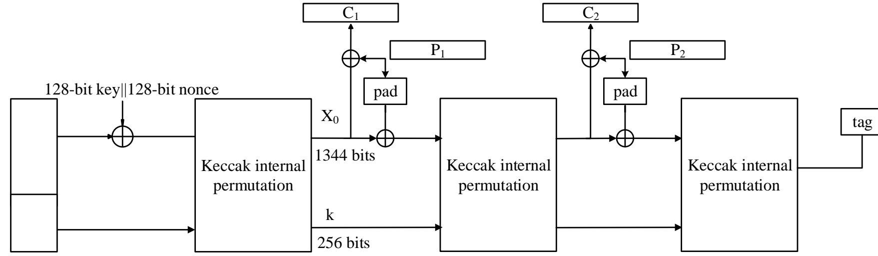


Fig. 3. Construction of Keyak on two blocks

2.3 Cube Tester.

Cube tester introduced in [18] is a distinguisher to detect algebraic property of cryptographic primitives. The idea is to reveal non-random behaviour of a Boolean function with algebraic degree d by summing its values when cube variables of size k ($k \leq d$) run over all of their 2^k inputs. The cube sum can be taken as higher order derivative [19] of the output polynomial with respect to cube variables. More precisely, we have

Theorem 1. *([10]) Given a polynomial $f : \{0,1\}^n \rightarrow \{0,1\}$ of degree d. Suppose that $0 < k \leq d$ and t is the monomial $\prod_{i=0}^{k-1} x_i$. Write f as:*

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

where none of the monomials in $Q_t(X)$ is divisible by t. Then the sum of f over all values of the cube (cube sum) is

$$\sum_{x' \in C_t} f(x', x_k, \dots, x_{n-1}) = P_t(x_k, \dots, x_{n-1}),$$

where the cube C_t contains all binary vectors of the length k.

Some properties for the polynomial P_t, such as its low algebraic degree and highly unbalanced truth table, have been extensively considered in [18] and [20].

$(n+1)$-Round Cube Tester on Keccak Sponge Functions. A cube tester can be constructed based on algebraic properties of Keccak sponge function to distinguish a round-reduced Keccak from a random permutation. An adversary can easily select a combination of $2^n + 1$ cube variables such that they are not multiplied with each other after the first round of Keccak. Note that after n-round Keccak the degree of these cube variables is at most 2^n. So the adversary can sum the output values over a cube of dimension $2^n + 1$ to get zero for a $(n + 1)$-round Keccak. This property is also used to perform MAC forgery attack in [10] when $n \leq 6$.

3 Conditional Cube Tester for Keccak Sponge Function

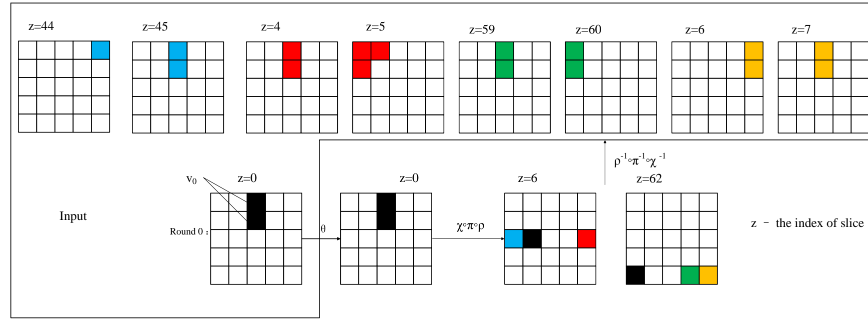


Fig. 4. Overview of bit conditions

As stated in Section 2, the cube attacks against the keyed modes of Keccak in [10] is to select the cube variables that are not multiplied with each other after the first round. Actually, it can be done simply in the context of the differential propagation. Let us consider the following example. In Figure 4, $A[2][0][0] = A[2][1][0] = v_0$ is set to be a cube variable and it only impacts two bits before the operation χ in the first round. To find $2^n + 1(n \leq 6)$ cube variables to construct an attack, one just needs to trace the positions of these bits.

In our new model, we develop a strategy to carefully choose the cube variables such that they are either not multiplied with each other after the second round or multiplied within a restrict set of variables.

The idea of our new model—the *conditional cube tester*, is to attach some bit conditions to a cube tester. Fig. 4 illustrates how to formulate such conditions. A detailed discussion will be given later in this section. To minimize the possibilities that the cube variable v_0 gets multiplied with other cube variables, we need to slow down the propagation of v_0. This can be done by imposing some additional conditions on the input message so that the coloured input bits of the second round are not related to v_0. Thus these coloured input bits of the second round

will not diffuse to other bits in the next round Keccak internal permutation. This is how the propagation of v_0 is controlled.

In the rest of this section, we shall define some types of cube variables in the CP kernel that are involved in the conditional cube tester. An important type is a set of variables that are well behaved through two rounds of Keccak, and we will see that some extra conditions on bits must be satisfied in order to get such variables. Then we will prove a useful result for these cube variables in a conditional cube tester. In the last part of the section, we shall discuss some properties on Keccak sponge function and describe algorithms based on these properties to examine multiplication relation after the second round between every pair of cube variables.

3.1 Conditional and Ordinary Cube Variables in a Conditional Cube Tester

In our discussion, cube variables are the variables in the CP kernel that are not multiplied with each other after the first round of Keccak. Now let us define two types of cube variables for the conditional cube tester.

Definition 1 *Cube variables that have propagation controlled in the first round and are not multiplied with each other after the second round of Keccak are called* ***conditional cube variables****. Cube variables that are not multiplied with each other after the first round and are not multiplied with any conditional cube variable after the second round are called* ***ordinary cube variables****.*

An ordinary cube variable has the advantage that it does not need to no extra conditions. However, there are no mechanisms to prevent ordinary cube variables from being multiplied with each other after the second round. Thus, in order to get an optimal cube tester for Keccak sponge function, a proper combinations of ordinary cube variables and conditional cube variables should be carefully selected.

To construct an $(n+2)$-round cube tester, we need to choose p conditional cube variables and q ordinary cube variables. With an appropriate choice of p and q, we have

Theorem 2. *For $(n+2)$-round Keccak sponge function $(n > 0)$, if there are p $(0 \leq p < 2^n + 1)$ conditional cube variables $v_1, \ldots, v_p$, and $q = 2^{n+1} - 2p + 1$ ordinary cube variables, $u_1, \ldots, u_q$ (If $q = 0$, we set $p = 2^n + 1$), then the term $v_1 v_2 \ldots v_p u_1 \ldots u_q$ will not appear in the output polynomials of $(n+2)$-round Keccak sponge function.*

Proof. Let $X_1, \cdots, X_s$ be the terms that contain v_i $(i = 1, \ldots, p)$ after the second round. Then by the definition of conditional variables, the degree of each X_j is one with respect to some $v_i (i = 1, \ldots, p)$. Similarly, let $Y_1, \cdots, Y_t$ be the terms that contain u_i $(i = 1, \ldots, q)$ after the second round. Then by the definition of ordinary variables, the degree of each Y_j is at most two with respect to some u_is $(i = 1, \ldots, p)$, and no v_i $(i = 1, \ldots, p)$ appears in Y_j.

For output polynomials after another n-round operation, a term with the highest degree with respect to $v_1, \dots, v_p$ and $u_1, \dots, u_q$ must be of the following form

$$T_{n+2} = X_{i_1} X_{i_2} \dots X_{i_k} Y_{j_1} Y_{j_2} \dots Y_{j_h} \quad \text{with } k + h = 2^n.$$

This implies that there are at most k distinct v_i and $2h$ distinct u_j can appear in T_{n+2}.

If T_{n+2} is divisible by $v_1 v_2 \dots v_p u_1 \dots u_q$, then we would have $k \geq p, 2h \geq q+1$ (since q is odd). This yields

$$k + h \geq p + \frac{q+1}{2} = p + 2^n - p + 1 > 2^n,$$

and we have reached a contradiction. □

Let us make some remarks on this theorem. The case that there is no conditional cube variable (i.e., $p = 0$) has been discussed extensively in [10], such as forgery attacks on Keccak-MAC and Keyak. For the case where $1 \leq p \leq 2^n + 1$, we can apply the conditional cube tester to recover the key for the $(n+2)$-round keyed modes of Keccak based on Theorem 2. The specific methods will be described in Section 4 and 5. Furthermore, in Section 6, we are able to use the case $p = 2^n + 1$ to implement the distinguishing attacks on Keccak sponge function.

In this paper, we only consider the cases when $n = 3, 4, 5$. If a proper combination of cube variables could be found for $n > 5$, the conditional cube tester still works.

3.2 Properties of Keccak Sponge Function

Before stating three useful properties of Keccak sponge function, we will describe the bitwise derivative of Boolean functions–a tool that helps us to explain our ideas accurately. The bitwise derivative of Boolean functions was proposed by Bo Zhu et.al and used to analyse Boolean algebra based block ciphers [21]. We observe that there is an equivalent relation between the differential characteristic and the bitwise derivatives of Boolean functions. However, it is much more efficient to trace the propagation of a variable by observing the differential characteristic rather than by computing the exact bitwise derivatives of Boolean functions. The bitwise derivative of a Boolean function is defined as follows.

Definition 2 *Given a Boolean function $f(x_0, x_1, \dots, x_{n-1})$, the bitwise derivative of f with respect to the variable x_m is defined as*

$$\delta_{x_m} f = f_{x_m=1} + f_{x_m=0}$$

The 0-th bitwise derivative is defined to be f itself. The i-th, where $i \geq 2$, bitwise derivative with respect to the variable sequence $(x_{m_1}, \dots, x_{m_i})$ is defined as

$$\delta^{(i)}_{x_{m_1},\dots,x_{m_i}} f = \delta_{x_{m_i}} (\delta^{(i-1)}_{x_{m_1},\dots,x_{m_{i-1}}} f)$$

Now let us describe differential properties of χ in the view of bitwise derivative. In this section, we first fix some notations. We will write the input of χ to be the (vector-valued) Boolean function $F = (f_0, f_1, f_2, f_3, f_4)$. The corresponding output is written as the (vector-valued) Boolean function $G = (g_0, g_1, g_2, g_3, g_4)$. The bitwise derivative of a (vector-valued) Boolean function is defined to be the (vector-valued) Boolean function by taking bitwise derivative in a componentwise manner.

Property 1. (**Bit Conditions**) If $\delta_{v_0} F = (1, 0, 0, 0, 0)$, then $\delta_{v_0} G = (1, 0, 0, 0, 0)$ if and only if $f_1 = 0$ and $f_4 + 1 = 0$.

Proof. By the structure of χ, the algebraic representation of the output Boolean function G is given by the following equations:

$$\begin{aligned} g_0 &= f_0 + (f_1 + 1)f_2, \\ g_1 &= f_1 + (f_2 + 1)f_3, \\ g_2 &= f_2 + (f_3 + 1)f_4, \\ g_3 &= f_3 + (f_4 + 1)f_0, \\ g_4 &= f_4 + (f_0 + 1)f_1. \end{aligned}$$

From the definition of the bitwise derivative, it can be deduced that $\delta_{v_0} G = (1, 0, 0, 0, f_4 + 1, f_1)$. It is clear that $\delta_{v_0} G = (1, 0, 0, 0, 0)$ if and only if $f_1 = 0$ and $f_4 + 1 = 0$. □

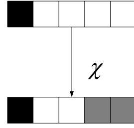


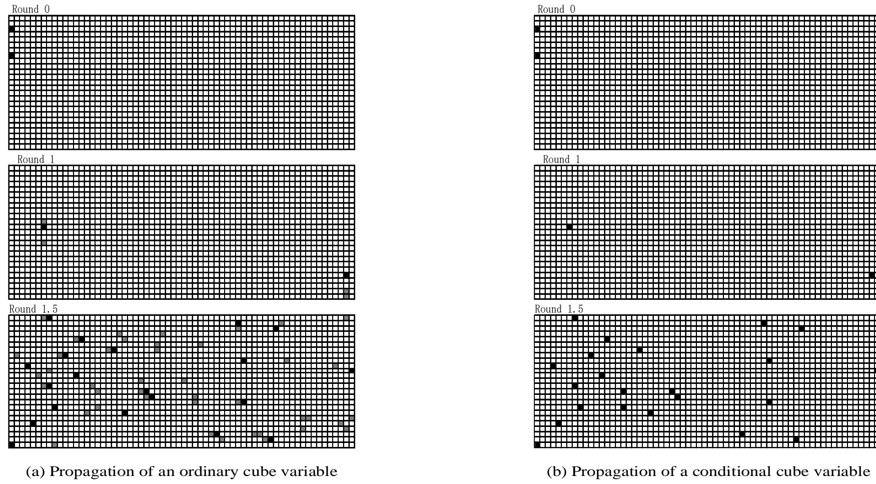
Fig. 5. Diffusion caused by operation χ

Now we explain the equivalence between the truncated differential characteristic and the bitwise derivatives of Boolean functions when tracing the propagation of a variable by using Fig. 5. Let the input difference for χ be $(1, 0, 0, 0, 0)$ and the truncated output difference is $(1, 0, 0, ?, ?)$ with '?' meaning an unknown bit. From the view of Boolean functions, the output vector $(1, 0, 0, ?, ?)$ indicates that $\delta_{v_0} g_0 = 1$, $\delta_{v_0} g_1 = 0$, $\delta_{v_0} g_2 = 0$ and both $\delta_{v_0} g_3, \delta_{v_0} g_4$ are some Boolean functions. From the view of the differential characteristic, if $f_1 = 0$ and $f_4 + 1 = 0$, then the differential characteristic $(1, 0, 0, 0, 0) \to (1, 0, 0, 0, 0)$ holds with probability 1. This also implies that g_0 is related to v_0 but g_i (for $1 \le i \le 4$) are independent of v_0. Therefore, the truncated differential characteristics and the bitwise derivatives of Boolean functions are equivalent representations.

Input/Output Bitwise Derivative(Difference)	Conditions
$(1,0,0,0,0) \rightarrow (1,0,0,0,0)$	$f_1 = 0, f_4 = 1$
$(0,1,0,0,0) \rightarrow (0,1,0,0,0)$	$f_2 = 0, f_0 = 1$
$(0,0,1,0,0) \rightarrow (0,0,1,0,0)$	$f_3 = 0, f_1 = 1$
$(0,0,0,1,0) \rightarrow (0,0,0,1,0)$	$f_4 = 0, f_2 = 1$
$(0,0,0,0,1) \rightarrow (0,0,0,0,1)$	$f_0 = 0, f_3 = 1$

Table 4. Summary of conditions for bitwise derivative of χ

We summarize all of the five input bitwise derivative cases in Table 4 where each input bitwise derivative has only one non-zero bit. Each case can be proved in a similar manner as Property 1. As discussed before, in each case, the input and output have the same vector of bitwise derivatives so that the propagation of v_0 by χ is under control. This will be used in constructing our conditional cube tester.



(a) Propagation of an ordinary cube variable

(b) Propagation of a conditional cube variable

Fig. 6. 1.5-round differential of an ordinary and a conditional cube variable

In order to show the advantage of a conditional cube variable over an ordinary cube variable, we consider the propagation of variable $A[2][0][0] = A[2][1][0] = v_0$ in two views: as an ordinary cube variable in the view of truncated differential characteristic (Fig. 6(a)) and as a conditional cube variable in the view of differential characteristic (Fig. 6(b)).

It is obvious to see that the two active bits at the beginning of the second round will affect 22 bits caused by the step θ. Thus, the conditional cube variable in Fig. 6(b) only relates to 22 active bits after 1.5-round Keccak internal permutation. However, not only bits with black colour but also those with gray colour after 1.5-round Keccak involve the ordinary cube variable in Fig. 6(a). In

total, there are 62 bits related to v_0 after 1.5-round Keccak. So it is more likely for a ordinary cube variable to get multiplied with other cube variables after the second round Keccak.

The pattern of the conditional cube variable v_0 in Fig. 6(b) will be called a 2-2-22 pattern to reflect the number of active bits in three states (the input state, the output state of the first round and the output state of the first 1.5 rounds).

During the process of searching more cube variables, we need to determine whether candidate variables get multiplied after the second round of Keccak and eliminate conditional cube variable candidates that require conflicting conditions. We observe that the following two properties with respect to the operation χ will be useful in dealing with these situations.

Property 2. (**Multiplication**) Assume that $\delta_{v_0}F = (\delta_{v_0}f_0, 0, 0, 0, 0)$ and $\delta_{v_1}F = (0, \delta_{v_1}f_1, 0, 0, 0)$ with $\delta_{v_0}f_0 \cdot \delta_{v_1}f_1 \neq 0$, then the term v_0v_1 will be in the output of χ.

Proof. As mentioned in the proof of the Property 1, the component g_4 of the output $G = (g_0, g_1, g_2, g_3, g_4)$ is $f_4 + (f_0 + 1)f_1$. From

$$\delta^{(2)}_{v_0,v_1}g_4 = \delta_{v_1}(\delta_{v_0}g_4) = \delta_{v_1}(\delta_{v_0}f_0) \cdot f_1 + \delta_{v_0}f_0 \cdot \delta_{v_1}f_1 = \delta_{v_0}f_0 \cdot \delta_{v_1}f_1$$

we see that $\delta^{(2)}_{v_0,v_1}g_4 \neq 0$ and hence g_4 contains the term v_0v_1. In particular, if $\delta_{v_0}f_0 = \delta_{v_1}f_1 = 1$, then $g_4 = v_0v_1 + h$, where h is a Boolean function not divisible by v_0v_1. □

Property 3. (**Exclusion**) If $\delta_{v_0}F = (1, 0, 0, 0, 0)$ and $\delta_{v_1}F = (0, 0, 1, 0, 0)$, then at least one of $\delta_{v_0}G = (1, 0, 0, 0, 0)$ and $\delta_{v_1}G = (0, 0, 1, 0, 0)$ is false.

Proof. From the Property 1 as well as the Table 4, the conditions $\delta_{v_0}F = (1, 0, 0, 0, 0)$ and $\delta_{v_0}G = (1, 0, 0, 0, 0)$ would imply $f_1 = 0, f_4 = 1$. Under the assumption $\delta_{v_1}F = (0, 0, 1, 0, 0)$, if $\delta_{v_1}G = (0, 0, 1, 0, 0)$ also holds true, then we would have $f_1 = 1, f_3 = 0$. This is a contradiction. □

For a version of Keccak sponge function, many positions in the plaintext space can be set as cube variables. For example, as shown in Fig. 7, we can set the bits in the same colour as a cube variable for the version Keccak-512. There are 256 such cases in 64 slices. Each of these cases in a version of Keccak is called a cube variable candidate.

Before searching for a proper combination of cube variables from these candidates to construct a conditional cube tester, we need to know the relation between every pair of cube variable candidates, namely, whether they are multiplied after the second round of Keccak. This problem could be solved directly by examining exact intermediate polynomials after the second round. However, it is very time-consuming to derive such an exact representation for the polynomials after the second round. Our approach with the application of truncated differentials can determine the (multiplication) relation between two cube variables

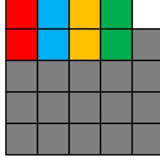


Fig. 7. Cube variable candidates in a slice for Keccak-512

efficiently. The precise procedures will be given in Algorithm 1, 2 and 3. These three algorithms are based on Property 2 and Property 3.

In the three algorithms, v_0 and v_1 are assumed to be two cube variable candidates in a Keccak version. We use $\delta_{v_0}A$ ($\delta_{v_1}A$) to denote the positions of v_0 (v_1) in the input state, which means to apply bitwise derivative on each entry of A. For example $\delta_{v_0}A[i][j][k] = 1$ means $A[i][j][k] = v_0 + h$, where h is a Boolean function independent of v_0. For a cube variable candidate v, we shall use '0', '1' and '2' to denote the inactive bit, the active bit and the unknown bit respectively. To be more specific, v is of type '0' if $\delta_v A[i][j][k] = 0$, v is of type '1' if $\delta_v A[i][j][k] = 1$ and type '2' if $\delta_v A[i][j][k]$ is a Boolean function. In this way, the truncated differences or differences in the algorithms can be used to interpret the bitwise derivatives on the state with respect to v.

Now we include three algorithms in this subsection for determining whether two possible cube variables (conditional or ordinary) have a multiplication relation after first round and the second round. The first algorithm is restricted to the case of two ordinary cube variable candidates. They should not be multiplied together after the first round.

Algorithm 1 Determine Relation of Two Ordinary Cube Variable Candidates

Input: $\delta_{v_0}A$ and $\delta_{v_1}A$ for two ordinary cube variable candidates v_0 and v_1
Output: multiplication relation of v_0 and v_1

```
1: compute the 0.5-round output difference B_0 (B_1) based on δ_{v_0}A(δ_{v_1}A);
2: flag=0
3: for each integer i ∈ [0,63], each integer j ∈ [0,4], each integer k ∈ [0,4] do
4:     if B_0[k][j][i] · B_1[k+1][j][i] = 1 then
5:         flag=1;        ▷ Property 2.
6:     end if
7: end for
8: if (flag) then
9:     return multiplied after the first round;
10: else
11:     return not multiplied after the first round;
12: end if
```

Algorithm 2 Determine Relation of a Conditional Cube Variable Candidate and an Ordinary Cube Variable Candidate

Input: $\delta_{v_0}A$ and $\delta_{v_1}A$ for the conditional cube variable candidate v_0 and the ordinary cube variable candidate v_1
Output: multiplication relation of v_0 and v_1

1: flag=[0,0]
2: compute the 0.5-round output difference B_0 (B_1) based on $\delta_{v_0}A(\delta_{v_1}A)$;
3: compute the 1.5-round truncated output difference C_0 (C_1) based on $\delta_{v_0}A(\delta_{v_1}A)$;
4: **for** each integer $i \in [0, 63]$, each integer $j \in [0, 4]$, each integer $k \in [0, 4]$ **do**
5:   **if** $B_0[k][j][i] \cdot B_1[k+1][j][i] = 1$ **then**
6:     flag[0]=1; ▷ Property 2.
7:   **end if**
8:   **if** $C_0[k][j][i] \cdot C_1[k+1][j][i] \neq 0$ **then**
9:     flag[1]=1; ▷ Property 2.
10:   **end if**
11: **end for**
12: **if** (flag[0]) **then**
13:   **return** multiplied after the first found;
14: **else if** (flag[1]) **then**
15:   **return** multiplied after the second round;
16: **end if**
17: **return** not multiplied after the second round;

Algorithm 3 Determine Relation of Two Conditional Cube Variable Candidates

Input: $\delta_{v_0}A$ and $\delta_{v_1}A$ for two conditional cube variable candidates v_0 and v_1
Output: multiplication relation of v_0 and v_1

1: flag=[0,0,0]
2: compute the 0.5-round output difference B_0 (B_1) based on $\delta_{v_0}A(\delta_{v_1}A)$;
3: compute the 1.5-round output difference C_0 (C_1) based on $\delta_{v_0}A(\delta_{v_1}A)$;
4: **for** each integer $i \in [0, 63]$, each integer $j \in [0, 4]$, each integer $k \in [0, 4]$ **do**
5:   **if** $B_0[k][j][i] \cdot B_1[k+2][j][i] = 1$ **then**
6:     flag[0]=1; ▷ Property 3.
7:   **end if**
8:   **if** $B_0[k][j][i] \cdot B_1[k+1][j][i] = 1$ **then**
9:     flag[1]=1; ▷ Property 2.
10:   **end if**
11:   **if** $C_0[k][j][i] \cdot C_1[k+1][j][i] = 1$ **then**
12:     flag[2]=1; ▷ Property 2.
13:   **end if**
14: **end for**
15: **if** (flag[0]) **then**
16:   **return** contradiction;
17: **else if** (flag[1]) **then**
18:   **return** multiplied after the first round;
19: **else if** (flag[2]) **then**
20:   **return** multiplied after the second round;
21: **end if**
22: **return** not multiplied by the second round;

The second algorithm is to test the relation between a conditional variable candidate and an ordinary cube variable candidate, whose multiplication is not allowed after the second round. The third algorithm is to examine the relation between two conditional variable candidates, whose multiplication is not allowed after the second round either.

4 Key Recovery Attack on Reduced-Round Keccak-MAC

In this section, we will use conditional cube testers to perform key recovery attacks against Keccak-MAC. First, we will discuss the general procedure for key recovery attack, including the attack process, complexity analysis and searching algorithm for suitable combinations of conditional and ordinary cube variables. Then we will describe conditional cube attacks to different variants of Keccak-MAC, including Keccak-MAC-512, Keccak-MAC-384 and Keccak-MAC-224.

4.1 General Process for Key Recovery Attack on Keccak-MAC

Given a cube tester with p conditional cube variables and $q = 2^{n+1} - 2p + 1$ ordinary cube variables ($1 \leq p \leq 2^n + 1$), we can construct a key recovery attack on $(n+2)$-round Keccak-MAC. In order to explain the general attack process clearly, we need to define some types of variables other than cube variables. As we know, a bit condition is an equality with a single variable on the left hand side and a Boolean function on the right hand side. The variable on the left hand side is called a **conditional variable**. Other public variables (that can be assigned to arbitrary values) are called **free variables**. Thus, a bit condition is a relation between conditional variable, equivalent key bit and free variables. It is assumed that s equivalent key bits are related to the bit conditions derived from conditional cube variables. The general attack process is described as follows.

Step 1. Assign free variables with random values.
Step 2. Guess values of the s equivalent key bits.
Step 3. Calculate the values of conditional variables under the guess of key bits.
Step 4. For each possible set of values of cube variables, compute the corresponding tag and then sum all of the 128-bit tags over the $(2^{n+1} - p + 1)$-dimension cube.
Step 5. If the sum is zero, the guess of these s key bits is probable correct and the process terminates; otherwise the guess is invalid, go back to Step 2.

After one execution of the above process, which takes $2^{2^{n+1}-p+1} \cdot 2^s$ Keccak calls at most, the values of s key bits can be recovered. To recover the remaining $128-s$ key bits, we just shift the positions of all the cube variables equally to the right along the z-direction and repeat the process for $128/s$ times. In this case, the bitwise derivatives with respect to the cube variables are rotated equally along the z axis as well. This rotation, known as translation invariance in the direction of the z axis, will change the equivalent key bits in the bit conditions but not the relations between the cube variables. Therefore, the time and data

complexity of the key recovery attack are both $\frac{1}{s} \cdot 2^{2^{n+1}-p+s+8} = \frac{2^{s-p}}{s} \cdot 2^{2^{n+1}+8}$. Thus, for an $(n+2)$-round conditional cube attack, the complexity is determined by $\frac{2^{s-p}}{s}$. We would like this term to be small to achieve a better performance. Notice that when the number of conditional cube variables gets larger, more key bits will be involved in the bit conditions and hence more guesses will be required. So p can not be too large to make the attack better. In our case, we use one conditional cube variable and $2^{n+1}-1$ ordinary cube variables to construct our key recovery attack on Keccak-MAC.

We choose $A[2][0][0] = A[2][1][0] = v_0$ as the conditional cube variable in our attacks. As shown in Fig.4, bit conditions are derived from $\delta_{v_0}A[2][0][6] = \delta_{v_0}A[2][4][6] = \delta_{v_0}A[4][3][62] = \delta_{v_0}A[4][4][62] = 0$, where A is the intermediate state after 1.5-round Keccak. This procedure could be done efficiently with the help of SAGE [22], a software on symbol computation. We fix $A[2][0][0] = A[2][1][0] = v_0$ as the conditional cube variable because there are only two equivalent key bits involved in the bit conditions. But if we choose other positions to set the conditional cube variable, the number of key bits involved in the bit conditions may be greater than two. Thus, $A[2][0][0] = A[2][1][0] = v_0$ is the cube variable and we find the corresponding ordinary cube variables using Algorithm 4.

Algorithm 4 Searching Ordinary Cube Variables along with the conditional cube variable $A[2][0][0] = A[2][1][0] = v_0$ for Keccak-MAC

Output: a set of ordinary cube variables;
1: m=#{ordinary cube variable candidates in bitrate part}
2: $S = \varnothing$
3: **for** each integer $i \in [0, m-1]$ **do**
4:   **execute** Algorithm 2 with v_0 and the i-th ordinary cube variable candidate u_i as the input;
5:   **if** Algorithm 2 returns 'not multiplied by the second round' **then**
6:     $S \leftarrow S \cup \{u_i\}$
7:   **end if**
8: **end for**
9: Choose the maximum number of variables in S which will not be multiplied with each other after the first round and put these variables into T
10: **return** T

In the discussion later, we will see that $2^{n+1}-1$ ordinary cube variables can be always found for $n = 3, 4$ and 5. So in these cases, the cube tester with v_0 and $2^{n+1}-1$ ordinary cube variables can be constructed to perform key recovery attacks on different variants of Keccak-MAC.

4.2 Key Recovery on 5/6/7-Round Keccak-MAC

We first discuss 5-round Keccak-MAC-512. In this case, $n = 3$ and full key bits can be recovered with one conditional cube variable and 15 ordinary cube

variables. The block size of this version is $1600 - 2 \cdot 512 = 576$ bits. As discussed in Section 4.1, we set $A[2][0][0] = A[2][1][0] = v_0$ to be the conditional cube variable. $A[4][0][44]$, $A[2][0][4]$, $A[2][0][59]$ and $A[2][0][27]$ are the conditional variables assigned with Boolean functions and a set of the corresponding ordinary cube variables is produced by Algorithm 4(see Table 5). To recover the remaining key bits, the positions of the conditional cube variable shall be shifted to $A[2][0][i] = A[2][1][i] = v_0(1 \leq i \leq 63)$ and the positions of ordinary cube variables shall be rotated at the same time. The key is recovered in 2^{24} time and data, which is very practical. On a desktop computer, the process of recovering a key only costs a few minutes.

The next example is a simple illustration of the attack where the key was generated randomly. For the convenience of statement, all the free variables are fixed to be zero, but they can be random bits. It is obvious that the correct key can be easily distinguished.

128-bit key:
11100001000101000000101101001000101111111000000110010111001110101
110001111000101111010001111111101000010101100000000110001001000010
correct value: $k_5 + k_{69} = 1, k_{60} = 0$
guessed value:00, cube sum: 0xe93169ae5c86d086, 0xf6ec898c859bea1a
guessed value:01, cube sum: 0xc7d0bc36dc141c5e, 0x523a33c8753eb171
guessed value:10, cube sum: 0x0,0x0
guessed value:11, cube sum: 0x2ee1d5988092ccd8, 0xa4d6ba44f0a55b6b

Ordinary Cube Variables	A[2][0][8]=A[2][1][8]=v_1, A[2][0][12]=A[2][1][12]=v_2, A[2][0][20]=A[2][1][20]=v_3, A[2][0][28]=A[2][1][28]=v_4, A[2][0][41]=A[2][1][41]=v_5, A[2][0][43]=A[2][1][43]=v_6, A[2][0][45]=A[2][1][45]=v_7, A[2][0][53]=A[2][1][53]=v_8, A[2][0][62]=A[2][1][62]=v_9, A[3][0][3]=A[3][1][3]=v_{10}, A[3][0][4]=A[3][1][4]=v_{11}, A[3][0][9]=A[3][1][9]=v_{12}, A[3][0][13]=A[3][1][13]=v_{13}, A[3][0][23]=A[3][1][23]=v_{14}, A[3][0][30]=A[3][1][30]=v_{15}
Conditional Cube Variables	A[2][0][0]=A[2][1][0]=v_0
Bit Conditions	A[4][0][44]=0, A[2][0][4]= k_5 + k_{69} + A[0][1][5] + A[2][1][4] + 1, A[2][0][59]= k_{60} + A[0][1][60] + A[2][1][59] + 1, A[2][0][7]= A[4][0][6] + A[2][1][7] + A[3][1][7]
Guessed Key Bits	$k_{60}, k_5 + k_{69}$

Table 5. Parameters set for attack on 5-round Keccak-MAC-512

To perform a conditional cube attack on 6-round Keccak-MAC-384, we use one conditional cube variable and 31 ordinary cube variables to recover full 128-bit key with 2^{40} Keccak calls. Fixing the conditional cube variable, we collect the corresponding ordinary cube variables by applying Algorithm 4. The parameters for this attack can be found in Table 9. It takes just a few days to run this attack on a desktop with four $i5$ processors. An instance for attacking 6-round Keccak-MAC-384 is summarized below, with randomly generated key and free variables are fixed to be zero:

128-bit key:

1111011111001001000111010010100111100011110001110111100100000010
0111000010010100010101110110111110100010101010001110111001100011
correct value: $k_5 + k_{69} = 1, k_{60} = 0$
guessed value:00, cube sum: 0x3f9d5fa4e143f779, 0x26607b3ce1c56f2b
guessed value:01, cube sum: 0x99bbf2ae6b93a7fb, 0xdbbb864fcc563747
guessed value:10, cube sum: 0x0,0x0
guessed value:11, cube sum: 0x398b37a846e81e42, 0x691cf4345e2164ee

For 7-round Keccak-MAC-256, our conditional cube attack takes 2^{72} Keccak calls to recover full 128-bit key, with a cube of dimension 64. We include the parameters of this attack in Table 10.

5 Key Recovery Attacks on Reduced-Round Keyak

Similar to the key recovery attack on Keyak in [10], we also deal with two-block messages (as depicted in Fig. 3) and allow the reuse of a nonce. In this way, we can use the first block to control the input of the second permutation and the second block to get the output of the second permutation. The attack described here is in fact a state recovery attack. We are able to get the bitrate part X_0 (see Fig. 3) but not the 256 bits in the capacity part. Denoting the capacity part as $k = (k_0, k_1, \cdots, k_{255})$, we will first recover k, then get the master key by performing the inverse of the first Keccak internal permutation.

In the attack, cube variables are set in the input state of the second Keccak internal permutation by choosing the values of P_1 while the second message block P_2 is set to zero bits. This implies that the second ciphertext block C_2 is the output of Keccak internal permutation. The attack procedure is almost identical to the general process described in Section 4.1 except for the bit conditions and the inverse process on the output. For 1344 output bits of Keyak, the operation χ of the last round on the most significant 1280 bits can be reversed. Note that the linear operations of the final round do not increase the degree of output polynomials, so the previous $(n+2)$-round cube tester can be used for $(n+3)$-round. In other words, conditional cube attack can be extended by one more round forward without increasing the dimension of cube.

For 7-round Keyak, conditional cube attack is built with the same cube as in Table 9 except for a different set of bit conditions as shown in Table 6. Note that in Table 6 A denotes the input state to the second internal permutation. By shifting the positions of cube variables and repeating the attack for $192/4 = 48$ times, three lanes of secret values, i.e. $k_0, \ldots, k_{191}$, can be recovered with $2^{36} \cdot 48 = 2^{41.58}$ Keyak calls. The other lane of key bits can be recovered by changing the conditional cube variable to $A[3][0][i] = A[3][1][i] = v_0$ and a set of the corresponding ordinary cube variables could be produced similarly by Algorithm 4. Since only one key bit is involved in the bit conditions after recovering three lanes of secret values, the remaining lane of secret values can be identified with $2^{33} \cdot 2^6 = 2^{39}$ Keyak calls. In total, the time complexity to recover the full 128-bit master key is about 2^{42} Keyak calls.

For 8-round Keyak, cube variables in Table 10 and bit conditions in the Table 6 are used in the conditional cube attack. Using a similar analysis as that to 7-round Keyak, the data and time complexities for 8-round attack are 2^{74}.

Finally, we remark that the memory complexity for both attacks can be neglected.

Bit conditions for 8(7)-round Keyak	A[4][0][44]= k_{169} (+A[4][1][44]) + A[2][2][45] + A[3][2][45] + A[4][2][44] + A[2][3][45] + A[4][3][44], A[0][0][5]= k_{128} + A[1][0][5] + A[2][0][4] + A[0][1][5] + A[2][1][4] + A[0][2][5] + A[2][2][4] + A[0][3][5] + A[2][3][4] + A[0][4][5] + 1, A[0][0][60]= k_{56} + k_{183} + A[2][0][59] + A[0][1][60] + A[2][1][59] + A[0][2][60] + A[2][2][59] + A[0][3][60] + A[2][3][59]+ A[0][4][60] + 1, A[2][0][7]= k_{131} + A[4][0][6] + A[2][1][7] + A[3][1][7] + A[4][1][6] + A[2][2][7]+ A[4][2][6] + A[2][3][7] + A[4][3][6]
Guessed Key Bits	k_{169}, k_{128}, k_{56} + k_{183}, k_{131}

Table 6. Parameters for attacking 7-round and 8-round Keyak

6 Distinguishing Attacks on Keccak Sponge Function

In this section, conditional cube tester will be applied to establish distinguishing attacks on Keccak sponge function with practical complexity. By Theorem 2, if we use $2^n + 1$ conditional cube variables, the monomial containing these $2^n + 1$ conditional cube variables will not appear in the output polynomials of $(n+2)$-round Keccak sponge function. This means that the dimension of the cube to distinguish $(n+2)$-round Keccak is reduced to $2^n + 1$ from higher numbers reported in [10]. In some cases like Keccak-512 and Keccak-384, the distinguishing attacks could be extended one more round forward.

Our construction of the cube tester includes two parts:

- Find a combination of sufficiently many conditional cube variables;
- Derive the corresponding bit conditions for the chosen conditional cube variables.

6.1 Constructing Conditional Cube Tester with MILP

A mixed-integer linear programming (MILP) problem is a linear programming problem with some variables taking integer values. MILP has been used to find the best differential characteristic in [23]. In this section, we model the problem of finding a combination of sufficiently many conditional cube variables as an MILP problem.

In this new model, each conditional cube variable candidate is assigned with a variable x_i ($1 \leq i \leq m$) where x_i takes value from $\{0, 1\}$. The i-th conditional cube variable candidate is selected as a conditional cube variable if and only if

$x_i = 1$. To find sufficiently many conditional cube variables, we need to find an assignment $X = \{(x_1, x_2, \ldots, x_m)|x_i \in \{0,1\}, 1 \leq i \leq m\}$ of hamming weight larger than $2^n + 1$. From earlier analysis, we know that in some cases two conditional cube variable candidates can not be selected simultaneously. We will first generate such constraints in terms of X. The precise generation procedure is the following.

Algorithm 5 Generating Constraints on X

Input: m conditional cube variable candidates;
Output: A set F of constrains on X

```
 1: F = ∅
 2: for each integer i ∈ [1, m − 1] do
 3:    for each integer j ∈ [i + 1, m] do
 4:       excute Algorithm 3 on the i-th and j-th conditional cube variable candidates;
 5:       if Algorithm 3 does not return 'Not Multiplied after the Second Round' then
 6:          F ← F ∪ {x_i + x_j ≤ 1}
 7:       end if
 8:    end for
 9: end for
10: return F
```

With the constraint set F , the selection problem for conditional cube variables is modeled into a binary linear programming problem as follow:

$$\sum_{i=1}^{m} x_i \geq 2^n + 1$$
$$\text{s.t.} A_0 X \leq b, X = \{(x_1, x_2, \ldots, x_m)|x_i \in \{0,1\}, \quad 1 \leq i \leq m\}$$

where A_0 is a binary matrix and b a binary vector such that $A_0 X \leq b$ describes the constraint set F. Although MILP is proved to be NP-hard, our problem is a special (and small) instance and can be solved by the programming solver Gurobi Optimizer [24] based on branch and cut algorithm.

We can get a desired combination of conditional cube variables by solving the MILP problem. In the rest of the section, we will construct distinguishing attacks on Keccak sponge function by solving the MILP problems and deriving the corresponding conditions for these conditional cube variables.

6.2 Distinguishing Attack on Keccak-512 and Keccak-384

As depicted in Fig. 7, there are 4 conditional cube variables candidates in one slice for Keccak-512. There are total 256 such candidates in 64 slices. Applying Algorithm 5 to generate all of the constraints with respect to these 256 candidates and solving the problem with Gurobi Optimizer, we get a set of 9 conditional cube variables. The bit conditions can be derived directly from $\delta_{v_m} A[i][j][k] = 0 (0 \leq m \leq 9)$, where $\delta_{v_m} A[i][j][k] \neq 1$ and A is the 1.5-round intermediate state. We then construct a 5-round conditional cube tester. Note that the algebraic

degree of output polynomial of 5-round Keccak-512 is at most 8, the cube sum of 5-round Keccak-512 output is zero.

The most significant 320 bits of Keccak-512 output can be reversed so that the distinguishing attack can be extended one more round further without increasing the complexity. The time complexity for the distinguishing attack on 6-round Keccak-512 with the conditional cube tester is thus 2^9 Keccak calls and the data complexity is also 2^9. From the fact that a distinguishing attack on Keccak with the capacity c_1 also works on Keccak with the capacity c_2 with the same complexity as long as $c_1 > c_2$, this attack can also distinguish Keccak-224, Keccak-256 up to 5 rounds and Keccak-384 up to 6 rounds.

We can find a combination of 17 conditional cube variables for Keccak-384 and construct a 7-round conditional cube tester in a similar manner with a complexity of 2^{17}.

The conditions for these two conditional cube tester are shown in Table 11 and Table 12. We have verified both of these two conditional cube testers by experiments.

6.3 Distinguishing Attack on Keccak-224

For Keccak-224, the same process can be applied with the conditional cube variables candidates in a 2-2-22 pattern. But with 1536 conditional cube variable candidates, the searching problem becomes difficult to solve. So we turn to consider the conditional cube variable candidates in double kernel patterns. The bitwise derivatives of such a chosen variable are still invariant with respect to the operation θ in the second round.

Four differential characteristics in double kernel pattern are shown in Table 7 in hexadecimal format with '-' denoting zero. The rows labeled with δ_0 and δ_1 are the input difference of the first round and the second round respectively; $\delta_{1.5}$ is the output difference after 1.5-round Keccak. The first two differential characteristics can be found in [9] in 6-6-6 pattern and the other two are found using the method in [8] in 8-8-8 pattern. As an example, a conditional cube variable can be set as

$$A[0][0][0]=A[0][1][0]=A[2][1][30]=A[2][2][30]= A[1][0][63]=A[1][2][63]=v_0.$$

This variable only impacts 6 bits after 1.5 round, which reduces the possibilities for the conditional cube variables to multiply with each other. Because of translation invariance in the direction of the z axis, we have 256 conditional cube variable candidates to build the MILP problem by applying Algorithm 5. With Gurobi Optimizer, we can find a combination of 30 conditional cube variables. Three conditional cube variables in 2-2-22 pattern have been added to the combination to get 33 independent conditional cube variables. Refer to Table 13 for the list of the conditional cube variables. The bit conditions can be derived exactly from the conditional cube variables, but they will be listed in the auxiliary supporting material due to the space limitation. Thus, a 7-round cube tester on Keccak-224 is constructed.

NO	δ_i	Differential				
0	δ_0	---------------1	8---------------	----------------	----------------	----------------
		---------------1	----------------	--------4-------	----------------	----------------
		----------------	8---------------	--------4-------	----------------	----------------
		----------------	----------------	----------------	----------------	----------------
		----------------	----------------	----------------	----------------	----------------
0	δ_1	---------------1	----------------	-------------2--	----------------	----------------
		----------------	----------------	----------------	----------------	----------------
		---------------1	------1---------	----------------	----------------	----------------
		----------------	------1---------	-------------2--	----------------	----------------
		----------------	----------------	----------------	----------------	----------------
0	$\delta_{1.5}$	---------------1	----------------	----------------	----------------	----------------
		----------------	----------------	---------------8	-----------2----	----------------
		----------------	----------------	----------------	----------------	----------------
		----------------	----------------	----4-----------	---------1------	----------------
		--------------8-	----------------	----------------	----------------	----------------
1	δ_0	----------------	----------------	----------------	----------------	----------------
		-----------8----	----------------	----------------	---------------1	----------------
		----------------	------------8---	----------------	---------------1	----------------
		-----------8----	------------8---	----------------	----------------	----------------
		----------------	----------------	----------------	----------------	----------------
1	δ_1	----------------	----------------	----------------	----------------	----------------
		----------------	----------------	----------------	1---------------	----------------
		----------------	----------------	---------2------	----------------	----------------
		----------------	--8-------------	---------2------	----------------	----------------
		----------------	--8-------------	----------------	1---------------	----------------
1	$\delta_{1.5}$	----------------	----------------	--------------1-	----------------	----------------
		----------------	----------------	----------------	------1---------	----------------
		----------------	----------------	----------------	----------------	----------------
		----------------	----------------	----------------	-----1----------	--1-------------
		----------------	---8------------	----------------	----------------	-2--------------
2	δ_0	----------------	---------------1	----------------	----------------	------4---------
		--------------1-	----------------	-------4--------	----------------	----------------
		----------------	----------------	-------4--------	----------------	------4---------
		--------------1-	---------------1	----------------	----------------	----------------
		----------------	----------------	----------------	----------------	----------------
2	δ_1	----------------	----------------	------------2---	----------------	----------------
		----------------	----------------	----------------	----2-----------	----------------
		---------------2	-----1----------	----------------	----------------	----------------
		---------------2	-----1----------	----------------	----------------	----------------
		----------------	----------------	------------2---	----2-----------	----------------
2	$\delta_{1.5}$	----------------	----------------	----------------	----------------	----------------
		----------------	----------------	--------------1-	----------2-----	-------------4--
		----------------	----------------	----------------	----------------	----------------
		----------------	----------------	---4------------	----------------	------2---------
		-------------8--	------1---------	----------------	-----4----------	----------------
3	δ_0	----------------	---------------1	----------------	----------------	------4---------
		--------------8-	----------------	----------------	--1-------------	----------------
		--------------8-	---------------1	----------------	--1-------------	------4---------
		----------------	----------------	----------------	----------------	----------------
		----------------	----------------	----------------	----------------	----------------
3	δ_1	----------------	----------------	----------------	----------------	----------------
		----------------	----------------	-------------4--	----------------	----------------
		---------------2	----------------	------------2---	----------------	----------------
		---------------2	-----8----------	-------------4--	----------------	----------------
		----------------	-----8----------	------------2---	----------------	----------------
3	$\delta_{1.5}$	----------------	----------------	-1--------------	----------------	----------------
		----------------	----------------	--------------1-	---------1------	-------------4--
		----------------	-----------1----	----------------	----------------	----------------
		----------------	----------------	----------------	---------2------	----------------
		----------------	----------------	----------------	-----4----------	----2-----------

Table 7. Four differential characteristics in double kernel pattern

The time complexity of this distinguishing attack is 2^{33}. Memory complexity is negligible. This distinguishing attack can be performed on a desktop computer in several hours.

7 Conclusion

In this paper, we propose the conditional cube tester for Keccak sponge function with the advantage of having smaller dimensions compared to the previous cube tester in some cases. Our approach is based on a novel idea to add some conditions for certain cube variables, so that the multiplication between cube variables are under control after the second round of Keccak sponge function. More specifically, using a conditional cube tester to round-reduced Keccak-MAC and Keyak, our key recovery attacks are more efficient than the currently best known attacks according to the number of rounds or the complexity. Another application of our conditional cube tester is to construct distinguishing attacks on Keccak sponge function. Our distinguishing attacks are much faster and improve the existing attacks. Most of our attacks are very practical and implementations and experiments have been conducted on desktop computers. We should also remark that our proposed conditional cube testers may be used to analyse Keccak-like cryptosystems.

References

1. Bertoni Guido, Daemen Joan, Peeters Michaël, and Gilles Van Assche. Keccak Sponge Function Family Main Document. `http://Keccak.noekeon.org/Keccak-main-2.1.pdf`.
2. Itai Dinur, Orr Dunkelman, and Adi Shamir. Improved Practical Attacks on Round-Reduced Keccak. *Journal of Cryptology*, 27(2):183–209, 2014.
3. Morawiecki Pawel, Pieprzyk Josef, Srebrny Marian, and Straus Michal. Preimage Attacks on the Round-Reduced Keccak with the Aid of Differential Cryptanalysis. Cryptology ePrint Archive, Report 2013/561, 2013. `http://eprint.iacr.org/`.
4. Daniel J. Bernstein. Second Preimages for 6 (7 (8??)) Rounds of Keccak. *NIST mailing list (2010).*
5. Ming Duan and XueJia Lai. Improved Zero-sum Distinguisher for Full Round Keccak-f Permutation. *Chinese Science Bulletin*, 57(6):694–697, 2012.
6. Alexandre Duc, Jian Guo, Thomas Peyrin, and Lei Wei. Unaligned Rebound Attack: Application to Keccak. In Anne Canteaut, editor, *FSE*, volume 7549 of LNCS, pages 402–421. Springer, 2012.
7. Jérémy Jean and Ivica Nikolić. Internal Differential Boomerangs: Practical Analysis of the Round-Reduced Keccak-f Permutation. In Gregor Leander, editor, *FSE 2015*, volume 9054 of LNCS, pages 537–556. Springer, 2015.
8. 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*, volume 7107 of LNCS, pages 236–254. Springer, 2011.
9. Sourav Das and Willi Meier. Differential Biases in Reduced-Round Keccak. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology – AFRICACRYPT 2014*, volume 8469 of LNCS, pages 69–87. Springer, 2014.

10. Itai Dinur, Paweł Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michał 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*, volume 9056 of LNCS, pages 733–761. Springer, 2015.
11. Bertoni Guido, Daemen Joan, Peeters Michaël, and Gilles Van Assche. Keyak. `http://keyak.noekeon.org`.
12. Mostafa Taha and Patrick Schaumont. Differential Power Analysis of MAC-Keccak at Any Key-Length. In Kazuo Sakiyama and Masayuki Terada, editors, *IWSEC*, volume 8231 of LNCS, pages 68–82. Springer, 2013.
13. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Cryptanalysis of Ascon. In Kaisa Nyberg, editor, *CT-RSA*, volume 9056 of LNCS, pages 371–387. Springer, 2015.
14. Morawiecki Pawel, Pieprzyk Josef, Straus Michal, and Srebrny Marian. Applications of Key Recovery Cube-attack-like. Cryptology ePrint Archive, Report 2015/1009, 2015. `http://eprint.iacr.org/`.
15. Itai Dinur and Adi Shamir. Breaking Grain-128 with Dynamic Cube Attacks. In Antoine Joux, editor, *FSE,* volume 6733 of LNCS, pages 167–187. Springer, 2011.
16. Simon Knellwolf, Willi Meier, and María Naya-Plasencia. Conditional Differential Cryptanalysis of NLFSR-Based Cryptosystems. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, volume 6447 of LNCS, pages 130–145. Springer, 2010.
17. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. `http://competitions.cr.yp.to/caesar.html`.
18. Jean-Philippe Aumasson, Itai Dinur, Willi Meier, and Adi Shamir. Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium. In Orr Dunkelman, editor, *FSE*, volume 5665 of LNCS, pages 1–22. Springer, 2009.
19. Xuejia Lai. Higher Order Derivatives and Differential Cryptanalysis. In Richard E. Blahut, Daniel J. Costello, Ueli Maurer, and Thomas Mittelholzer, editors, *Communications and Cryptography: Two Sides of One Tapestry*, pages 227–233. Springer, 1994.
20. Itai Dinur and Adi Shamir. Cube Attacks on Tweakable Black Box Polynomials. In Antoine Joux, editor, *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of LNCS, pages 278–299. Springer, 2009.
21. Bo Zhu, Kefei Chen, and Xuejia Lai. Bitwise Higher Order Differential Cryptanalysis. In Liqun Chen and Moti Yung, editors, *INTRUST 2009*, volume 6163 of LNCS, pages 250–262. Springer, 2010.
22. William Stein and David Joyner. Sage: System for algebra and geometry experimentation. *ACM SIGSAM Bull.*, 39(2):61–64, June 2005.
23. Siwei Sun, Lei Hu, Meiqin Wang, Qianqian Yang, Kexin Qiao, Xiaoshuang Ma, Ling Song, and Jinyong Shan. Extending the Applicability of the Mixed-Integer Programming Technique in Automatic Differential Cryptanalysis. In Javier Lopez and J. Chris Mitchell, editors, *ISC*, volume 9290 of LNCS, pages 141–157. Springer, 2015.
24. Gurobi optimization. *Gurobi:Gurobi optimizer reference manual (2015)*. `http://www.gurobi.com`.

0	1	62	28	27
36	44	6	55	20
3	10	43	25	39
41	45	15	21	8
18	2	61	56	14

Table 8. offsets r[x,y] in operation ρ

Ordinary Cube Variables	A[2][0][12]=A[2][1][12]=v_1, A[2][0][20]=A[2][1][20]=v_2, A[2][0][28]=A[2][1][28]=v_3, A[2][0][41]=A[2][1][41]=v_4, A[2][0][43]=A[2][1][43]=v_5, A[2][0][45]=A[2][1][45]=v_6, A[2][0][53]=A[2][1][53]=v_7, A[2][0][61]=A[2][1][61]=v_8, A[2][0][62]=A[2][1][62]=v_9, A[3][0][3]=A[3][1][3]=v_{10}, A[3][0][9]=A[3][1][9]=v_{11}, A[3][0][13]=A[3][1][13]=v_{12}, A[3][0][15]=A[3][1][15]=v_{13}, A[3][0][23]=A[3][1][23]=v_{14}, A[3][0][30]=A[3][1][30]=v_{15}, A[3][0][40]=A[3][1][40]=v_{16}, A[3][0][46]=A[3][1][46]=v_{17}, A[3][0][56]=A[3][1][56]=v_{18}, A[3][0][57]=A[3][1][57]=v_{19}, A[4][0][5]=A[4][1][5]=v_{20}, A[4][0][10]=A[4][1][10]=v_{21}, A[4][0][12]=A[4][1][12]=v_{22}, A[4][0][14]=A[4][1][14]=v_{23}, A[4][0][47]=A[4][1][47]=v_{24}, A[4][0][58]=A[4][1][58]=v_{25}, A[4][0][62]=A[4][1][62]=v_{26}, A[4][0][63]=A[4][1][63]=v_{27}, A[0][1][28]=A[0][2][28]=v_{28}, A[0][1][34]=A[0][2][34]=v_{29}, A[0][1][37]=A[0][2][37]=v_{30}, A[0][1][46]=A[0][2][46]=v_{31}
Conditional Cube Variables	A[2][0][0]=A[2][1][0]=v_0
Bit Conditions	A[4][0][44]= A[4][1][44] + A[2][2][45], A[2][0][4]= k_5 + k_{69} + A[0][1][5] + A[2][1][4] + A[0][2][5] + A[2][2][4] + 1, A[2][0][59]= k_{60} + A[0][1][60] + A[2][1][59] + A[0][2][60] + A[2][2][59] + 1, A[2][0][7]= A[4][0][6] + A[2][1][7] + A[4][1][6] + A[2][2][7] + A[3][1][7].
Guessed Key Bits	$k_{60}, k_5 + k_{69}$

Table 9. Parameters set for attack on 6-round Keccak-MAC-384

Ordinary Cube Variables	A[2][0][8]=A[2][1][8]=v_1, A[2][0][12]=A[2][1][12]=v_2, A[2][0][20]=A[2][1][20]=v_3, A[2][0][28]=A[2][1][28]=v_4, A[2][0][41]=A[2][1][41]=v_5, A[2][0][43]=A[2][1][43]=v_6, A[2][0][45]=A[2][1][45]=v_7, A[2][0][53]=A[2][1][53]=v_8, A[2][0][62]=A[2][1][62]=v_9, A[3][0][3]=A[3][1][3]=v_{10}, A[3][0][9]=A[3][1][9]=v_{11}, A[3][0][13]=A[3][1][13]=v_{12}, A[3][0][30]=A[3][1][30]=v_{13}, A[3][0][40]=A[3][1][40]=v_{14}, A[3][0][46]=A[3][1][46]=v_{15}, A[3][0][56]=A[3][1][56]=v_{16}, A[4][0][5]=A[4][1][5]=v_{17}, A[4][0][10]=A[4][1][10]=v_{18}, A[4][0][12]=A[4][1][12]=v_{19}, A[4][0][14]=A[4][1][14]=v_{20}, A[4][0][31]=A[4][1][31]=v_{21}, A[4][0][47]=A[4][1][47]=v_{22}, A[4][0][58]=A[4][1][58]=v_{23}, A[4][0][62]=A[4][1][62]=v_{24}, A[4][0][63]=A[4][1][63]=v_{25}, A[0][1][37]=A[0][2][37]=v_{26}, A[0][1][47]=v_{27}, A[0][2][47]=v_{27}+v_{28}, A[0][3][47]=v_{28}, A[0][1][46]=A[0][2][46]=v_{29}, A[0][1][59]=A[0][2][59]=v_{30}, A[1][1][7]=A[1][2][7]=v_{31}, A[1][1][15]=A[1][2][15]=v_{32}, A[1][1][20]=A[1][2][20]=v_{33}, A[1][1][26]=A[1][2][26]=v_{34}, A[1][1][30]=A[1][2][30]=v_{35}, A[1][1][38]=A[1][2][38]=v_{36}, A[1][1][39]=A[1][2][39]=v_{37}, A[1][1][40]=A[1][2][40]=v_{38}, A[1][1][52]=A[1][2][52]=v_{39}, A[1][1][54]=A[1][2][54]=v_{40}, A[2][1][11]=A[2][2][11]=v_{41}, A[2][1][15]=A[2][2][15]=v_{42}, A[2][1][19]=A[2][2][19]=v_{43}, A[2][1][24]=A[2][2][24]=v_{44}, A[2][1][52]=A[2][2][52]=v_{45}, A[2][1][58]=A[2][2][58]=v_{46}, A[2][1][61]=A[2][2][61]=v_{47}, A[3][1][23]=A[3][2][23]=v_{48}, A[3][1][29]=A[3][2][29]=v_{49}, A[3][1][58]=A[3][2][58]=v_{50}, A[4][1][1]=A[4][2][1]=v_{51}, A[4][1][28]=A[4][2][28]=v_{52}, A[4][1][44]=A[4][2][44]=v_{53}, A[4][1][50]=A[4][2][50]=v_{54}, A[4][1][61]=A[4][2][61]=v_{55}, A[0][2][17]=A[0][3][17]=v_{56}, A[0][2][28]=A[0][3][28]=v_{57}, A[0][2][34]=A[0][3][34]=v_{58}, A[0][2][56]=A[0][3][56]=v_{59}, A[1][2][44]=A[1][3][44]=v_{60}, A[1][2][49]=A[1][3][49]=v_{61}, A[1][2][57]=A[1][3][57]=v_{62}, A[2][0][5]=A[2][2][5]=v_{63}
Conditional Cube Variables	A[2][0][0]=A[2][1][0]=v_0
Bit Conditions	A[4][0][44]= A[2][2][45] + A[3][2][45], A[2][0][4]= k_5 + k_{69} + A[0][1][5] + A[2][1][4] + A[0][2][5] + A[2][2][4] + A[0][3][5] + 1, A[2][0][59]= k_{60} + A[0][1][60] + A[2][1][59] + A[0][2][60] + A[2][2][59] + A[0][3][60] + 1, A[2][0][7]= A[3][1][7] + A[4][1][6] + A[2][1][7] + A[4][1][6] +A[2][2][7] + A[4][2][6]
Guessed Key Bits	$k_{60}, k_5 + k_{69}$

Table 10. Parameters set for attack on 7-round Keccak-MAC-256

Conditional Cube Variables	A[2][0][0]=A[2][1][0]=v_0, A[2][0][1]=A[2][1][1]=v_1,A[2][0][2]=A[2][1][2]=v_2, A[2][0][3]=A[2][1][3]=v_3,A[2][0][22]=A[2][1][22]=v_4, A[2][0][23]=A[2][1][23]=v_5, A[2][0][44]=A[2][1][44]=v_6, A[2][0][45]=A[2][1][45]=v_7,A[3][0][15]=A[3][1][15]=v_8
Bit Conditions	A[2][0][4]= A[0][0][5]+ A[1][0][5]+ A[0][1][5]+ A[2][1][4]+ 1, A[2][0][5]= A[0][0][6]+ A[1][0][6]+ A[0][1][6]+ A[2][1][5]+ 1, A[2][0][6]= A[0][0][7]+ A[1][0][7]+ A[0][1][7]+ A[2][1][6]+ 1, A[2][0][7]= A[0][0][8]+ A[1][0][8]+ A[0][1][8]+ A[2][1][7]+ 1, A[2][0][8]= A[4][0][7]+ A[2][1][8]+ A[3][1][8], A[2][0][9]= A[4][0][8]+ A[2][1][9]+ A[3][1][9], A[2][0][10]= A[4][0][9]+ A[2][1][10]+ A[3][1][10], A[2][0][17]= A[0][0][18]+ A[0][1][18]+ A[2][1][17]+ 1, A[2][0][25]= A[4][0][24]+ A[2][1][25], A[2][0][26]= A[0][0][27]+ A[1][0][27]+ A[0][1][27]+ A[2][1][26]+ 1, A[2][0][27]= A[0][0][28]+ A[1][0][28]+ A[0][1][28]+ A[2][1][27]+ 1, A[2][0][29]= A[4][0][28]+ A[2][1][29]+ A[3][1][29], A[2][0][30]= A[4][0][29]+ A[2][1][30]+ A[3][1][30], A[2][0][40]= A[0][0][41]+ A[0][1][41]+ A[2][1][40]+ 1, A[2][0][46]= A[4][0][45]+ A[2][1][46], A[2][0][47]= A[4][0][46]+ A[2][1][47], A[2][0][48]= A[4][0][47]+ A[2][1][48], A[2][0][49]= A[0][0][50]+ A[1][0][50]+ A[0][1][50]+ A[2][1][49]+ 1, A[2][0][51]= A[4][0][50]+ A[2][1][51]+ A[3][1][51], A[2][0][52]= A[4][0][51]+ A[2][1][52]+ A[3][1][52], A[2][0][59]= A[0][0][60]+ A[0][1][60]+ A[2][1][59]+ 1, A[2][0][60]= A[0][0][61]+ A[0][1][61]+ A[2][1][60]+ 1, A[2][0][61]= A[0][0][62]+ A[0][1][62]+ A[2][1][61]+ 1, A[2][0][62]= A[0][0][63]+ A[0][1][63]+ A[2][1][62]+ 1, A[3][0][23]= A[0][0][22]+ A[0][1][22]+ A[3][1][23], A[3][0][31]= A[0][0][30]+ A[0][1][30]+ A[3][1][31], A[3][0][45]= A[1][0][46]+ A[1][1][46]+ A[3][1][45]+ 1, A[4][0][3]= A[0][0][5]+ A[1][0][5]+ A[0][1][5]+ 1, A[4][0][6]= A[0][0][8]+ A[1][0][8]+ A[0][1][8]+ A[3][1][7]+ 1, A[4][0][25]= A[0][0][27]+ A[1][0][27]+ A[0][1][27]+ 1, A[0][1][49]= A[0][0][49]+ A[1][0][49]+ A[4][0][47]+ 1, A[4][0][44]=0, A[4][0][2] = 1.

Table 11. Conditions to distinguish Keccak-512

Conditional Cube Variables	A[0][0][14]=A[0][1][14]=v_0, A[2][0][23]=A[2][1][23]=v_1,A[2][0][24]=A[2][1][24]=v_2, A[2][0][43]=A[2][1][43]=v_3, A[2][0][44]=v_4, A[2][1][44]=v_4+v_5, A[2][2][44]=v_5, A[3][0][56]=A[3][1][56]=v_6, A[3][0][58]=A[3][1][58]=v_7, A[0][1][57]=A[0][2][57]=v_8, A[0][1][58]=A[0][2][58]=v_9, A[1][1][49]=A[1][2][49]=v_{10}, A[1][1][50]=A[1][2][50]=v_{11}, A[2][1][41]=A[2][2][41]=v_{12}, A[0][0][20]=A[0][2][20]=v_{13}, A[1][0][13]=A[1][2][13]=v_{14}, A[2][0][0]=A[2][2][0]=v_{15}, A[2][0][16]=A[2][2][16]=v_{16}
Bit Conditions	A[0][0][1]= A[3][0][2]+ A[0][1][1]+ A[3][1][2]+ A[4][1][2]+ A[0][2][1] A[0][0][2]= A[3][0][3]+ A[0][1][2]+ A[3][1][3]+ A[4][1][3]+ A[0][2][2]+ 1 A[0][0][5]= A[3][0][6]+ A[0][1][5]+ A[3][1][6]+ A[0][2][5]+ 1 A[0][0][7]= A[3][0][8]+ A[0][1][7]+ A[3][1][8]+ A[0][2][7] A[0][0][9]= A[3][0][10]+ A[0][1][9]+ A[3][1][10]+ A[0][2][9] A[0][0][12]= A[2][0][11]+ A[0][1][12]+ A[2][1][11]+ A[0][2][12]+ A[2][2][11]+ 1 A[0][0][15]= A[2][0][14]+ A[0][1][15]+ A[2][1][14]+ A[0][2][15]+ A[2][2][14] A[0][0][16]= A[2][0][15]+ A[0][1][16]+ A[2][1][15]+ A[0][2][16]+ A[2][2][15] A[0][0][19]= A[2][0][18]+ A[0][1][19]+ A[2][1][18]+ A[0][2][19]+ A[2][2][18]+ 1 A[0][0][22]= A[3][0][23]+ A[0][1][22]+ A[3][1][23]+ A[4][1][23]+ A[0][2][22]+ A[2][2][24]+ 1 A[0][0][28]= A[1][0][29]+ A[2][0][27]+ A[2][0][28] + A[4][0][29]+ A[0][1][28]+ A[0][1][29]+ A[1][1][28]+ A[2][1][27]+ A[2][1][28] + A[4][1][29]+ A[0][2][28]+ A[0][2][29]+ A[1][2][28]+ A[2][2][27]+ A[2][2][28]+ 1 A[0][0][29]= A[1][0][29]+ A[2][0][28]+ A[0][1][29]+ A[2][1][28]+ A[0][2][29]+ A[2][2][28]+ 1 A[0][0][30]= A[1][0][29]+ A[4][0][30]+ A[1][1][29]+ A[4][1][30]+ A[1][2][29]+ 1 A[0][0][34]= A[2][0][33]+ A[0][1][34]+ A[1][1][34]+ A[2][1][33]+ A[0][2][34]+ A[2][2][33] A[0][0][39]= A[4][0][37]+ A[0][1][39]+ A[4][1][37]+ A[0][2][39]+ 1 A[0][0][40]= A[3][0][41]+ A[0][1][40]+ A[3][1][41]+ A[4][1][41]+ A[0][2][40]+ 1 A[0][0][42]= A[2][0][41]+ A[0][1][42]+ A[0][2][42] A[0][0][43]= A[2][0][42]+ A[0][1][43]+ A[1][1][43]+ A[2][1][42]+ A[0][2][43]+ A[2][2][42]+ 1 A[0][0][46]= A[1][2][46]+ A[2][0][45]+ A[0][1][46]+ A[2][1][45]+ A[0][2][46]+ A[2][2][45]+ 1 A[0][0][48]= A[1][0][48]+ A[2][0][47]+ A[0][1][48]+ A[2][1][47]+ A[0][2][48]+ A[2][2][47]+ 1 A[0][0][49]= A[2][0][48]+ A[2][0][49]+ A[3][0][48]+ A[0][1][49]+ A[2][1][48] + A[3][1][48]+ A[0][2][49]+ A[2][2][48] A[0][0][60]= A[2][0][59]+ A[0][1][60]+ A[2][1][59]+ A[0][1][60]+ A[2][2][59]+ 1 A[0][0][63]= A[3][0][0]+ A[0][1][63]+ A[3][1][0]+ A[0][2][63]+ 1 A[1][0][8]= A[3][0][7]+ A[1][1][8]+ A[3][1][7]+ A[1][2][8] A[1][0][22]= A[0][1][23]+ A[1][1][22]+ A[1][2][22]+ A[2][2][24]+ 1 A[1][0][23]= A[4][0][24]+ A[0][1][24]+ A[1][1][23]+ A[4][1][24]+ A[1][2][23]+ 1 A[1][0][25]= A[3][0][24]+ A[1][1][25]+ A[3][1][24]+ A[1][2][25]+ 1 A[1][0][28]= A[1][0][29]+ A[2][0][28]+ A[4][0][29]+ A[0][1][29]+ A[1][1][28]+ A[2][1][28]+ A[4][1][29]+ A[0][2][29]+ A[1][2][28]+ A[2][2][28] A[1][0][44]= A[3][0][43]+ A[1][1][44]+ A[3][1][43]+ A[1][2][44] A[1][0][45]= A[3][0][44]+ A[1][1][45]+ A[3][1][44]+ A[1][2][45] A[1][0][49]= A[2][0][49]+ A[3][0][48]+ A[3][1][48]+ 1 A[1][0][50]= A[4][0][51]+ A[0][1][51]+ A[4][4][51]+ 1 A[1][0][51]= A[3][0][50]+ A[1][1][51]+ A[3][1][50]+ A[1][2][51]+ A[2][2][51] A[1][0][59]= A[4][0][60]+ A[1][1][59]+ A[4][1][60]+ A[1][2][59] A[2][0][2]= A[4][0][1]+ A[2][1][2]+ A[4][1][1]+ A[2][2][2] A[2][0][4]= A[2][1][4]+ A[2][2][4] A[2][0][5]= A[4][0][4]+ A[2][1][5]+ A[4][1][4]+ A[2][2][5] A[2][0][7]= A[4][0][6]+ A[2][1][7]+ A[3][1][7]+ A[4][1][6]+ A[2][2][7] A[2][0][22]= A[4][0][21]+ A[2][1][22]+ A[4][1][21]+ A[2][2][22] A[2][0][25]= A[4][0][24]+ A[2][1][25]+ A[4][0][24]+ A[2][2][25] A[2][0][30]= A[4][0][29]+ A[2][1][30]+ A[3][1][30]+ A[4][1][29]+ A[2][2][30] A[2][0][31]= A[4][0][30]+ A[2][1][31]+ A[3][1][31]+ A[4][1][30]+ A[2][2][31] A[2][0][38]= A[4][0][37]+ A[2][1][38]+ A[4][1][37]+ A[2][2][38] A[2][0][39]= A[3][0][41]+ A[2][1][39]+ A[3][1][41]+ A[4][1][41]+ A[2][2][39] A[2][0][50]= A[4][0][49]+ A[2][1][50]+ A[3][1][50]+ A[4][1][49]+ A[2][2][50] A[2][0][51]= A[2][2][51]+ 1 A[2][0][62]= A[3][0][0]+ A[1][1][63]+ A[2][1][62]+ A[3][1][0]+ A[2][2][62] A[2][0][63]= A[4][0][62]+ A[2][1][63]+ A[4][1][62]+ A[2][2][63] A[3][0][22]= A[4][0][24]+ A[0][1][24]+ A[3][1][22]+ A[4][1][24] A[3][0][40]= A[4][0][37]+ A[3][1][40]+ A[4][1][37]+ A[4][1][40] A[3][0][49]= A[4][0][51]+ A[0][1][51]+ A[3][1][49]+ A[4][1][51]+ A[2][2][50]+ 1 A[4][0][2]= A[4][1][2]+ 1 A[4][0][22]= A[3][1][23]+ A[4][1][22]+ A[2][2][23] A[4][0][23]= A[4][1][23]+ A[2][2][24] A[4][0][50]= A[2][1][51]+ A[3][1][51]+ A[4][1][50]+ 1 A[0][1][20]= A[2][0][19]+ A[2][1][19]+ A[2][2][19]+ 1 A[1][2][40]= 1 A[4][1][0]= 1 A[1][2][19]= 1 A[1][2][20]= 1 A[1][1][40]= 1 A[2][1][8]= 0 A[1][1][15]= 1

Table 12. Conditions to Distinguish Keccak-384

Conditional Cube Variables
A[0][0][3]=A[1][2][2]=v_0, A[0][1][3]=A[2][1][33]=A[2][2][33]=v_0+v_{25}, A[1][0][2]=v_0+v_{17},
A[0][0][6]=A[1][0][5]=A[1][2][5]=v_1, A[0][1][6]=A[2][1][36]=A[2][2][36]=v_1+v_{17},
A[0][0][9]=A[0][1][9]=A[2][1][39]=A[2][2][39]=A[1][0][8]=A[1][2][8]=v_2,
A[0][0][11]=A[1][0][10]=v_3, A[0][1][11]=A[2][1][41]=A[2][2][41]=v_3+v_{18}, A[1][2][10]=v_3+v_{16},
A[0][0][14]=A[2][1][44]=A[2][2][44]=A[1][0][13]=A[1][2][13]=v_4, A[0][1][14]=$v_4+v_{16}+v_{26}$,
A[0][0][16]=v_5, A[0][1][16]=A[2][1][46]=A[2][2][46]=v_5+v_{19}, A[1][0][15]=$v_5+v_{20}+v_{27}$, A[1][2][15]=v_5+v_{27},
A[0][0][19]=A[1][0][18]=A[1][2][18]=v_6, A[0][1][19]=A[2][1][49]=A[2][2][49]=v_6+v_{20},
A[0][0][21]=A[0][1][21]=A[2][1][51]=A[2][2][51]=v_7, A[1][0][20]=$v_7+v_{21}+v_{28}$, A[1][2][20]=v_7+v_{28},
A[0][0][22]=A[2][1][52]=A[2][2][52]=A[1][0][21]=v_8, A[0][1][22]=A[1][2][21]=v_8+v_{14},
A[0][0][24]=A[1][0][23]=A[1][2][23]=v_9, A[0][1][24]=A[2][1][54]=A[2][2][54]=v_9+v_{21},
A[0][0][27]=A[2][1][57]=A[2][2][57]=A[1][0][26]=A[1][2][26]=v_{10}, A[0][1][27]=$v_{10}+v_{28}$,
A[0][0][29]=A[1][0][28]=v_{11}, A[0][1][29]=A[2][1][59]=A[2][2][59]=$v_{11}+v_{22}$, A[1][2][28]=$v_{11}+v_{15}$,
A[0][1][32]=$v_{12}+v_{15}+v_{29}$, A[0][0][32]=A[2][1][62]=A[2][2][62]=A[1][0][31]=A[1][2][31]=v_{12},
A[0][0][62]=A[1][2][61]=v_{13}, A[0][1][62]=A[2][1][28]=A[2][2][28]=$v_{13}+v_{23}$, A[1][0][61]=$v_{13}+v_{24}$,
A[3][1][6]=A[3][2][6]=A[1][3][21]=A[0][1][25]=A[0][3][25]=v_{14},
A[3][1][13]=A[3][2][13]=$v_{15}+v_{29}$, A[1][3][28]=A[0][3][32]=v_{15},
A[3][1][59]=A[3][2][59]=$v_{16}+v_{26}$, A[1][3][10]=A[0][3][14]=v_{16}
A[1][3][2]=A[4][0][40]=A[4][2][40]=A[0][3][6]=v_{17},
A[1][0][7]=A[4][0][45]=A[4][2][45]=$v_{18}+v_{26}$, A[1][3][7]=A[0][3][11]=v_{18}
A[1][0][12]=A[1][3][12]=A[4][0][50]=A[4][2][50]=A[0][3][16]=v_{19},
A[1][3][15]=A[0][3][19]=v_{20}, A[4][0][53]=A[4][2][53]=$v_{20}+v_{27}$,
A[1][3][20]=A[0][3][24]=v_{21}, A[4][0][58]=A[4][2][58]=$v_{21}+v_{28}$,
A[1][0][25]=A[4][0][63]=A[4][2][63]=$v_{22}+v_{29}$, A[1][3][25]=A[0][3][29]=v_{22},
A[1][0][58]=A[1][3][58]=A[4][0][32]=A[4][2][32]=A[0][3][62]=v_{23},
A[1][3][61]=A[4][0][35]=A[4][2][35]=A[0][1][1]=A[0][3][1]=A[2][1][31]=A[2][2][31]=v_{24},
A[1][0][63]=A[1][3][63]=A[4][0][37]=A[4][2][37]=A[0][3][3]=v_{25},
A[1][2][7]=A[0][2][14]=v_{26}, A[0][2][22]=A[3][1][3]=A[3][2][3]=v_{27}
A[0][2][27]=A[3][1][8]=A[3][2][8]=v_{28}, A[1][2][25]=A[0][2][32]=v_{29}, A[2][0][55]=A[2][1][55]=v_{30}
A[0][2][60]=A[0][3][60]=v_{31}, A[0][1][37]=A[0][3][37]=v_{32}

Table 13. Conditional Cube Variables to Distinguish Keccak-224