Practical Complexity Cube Attacks on Round-Reduced Keccak Sponge Function

Itai Dinur[1], Paweł Morawiecki[2,3], Josef Pieprzyk[4] Marian Srebrny[2,3], and Michał Straus[3]

[1] Computer Science Department, École Normale Supérieure, France
[2] Institute of Computer Science, Polish Academy of Sciences, Poland
[3] Section of Informatics, University of Commerce, Kielce, Poland
[4] Queensland University of Technology, Brisbane, Australia

Abstract. In this paper we mount the cube attack on the Keccak sponge function. The cube attack, formally introduced in 2008, is an algebraic technique applicable to cryptographic primitives whose output can be described as a low-degree polynomial in the input. Our results show that 5- and 6-round Keccak sponge function is vulnerable to this technique. All the presented attacks have practical complexities and were verified on a desktop PC.

Keywords: Keccak, SHA-3, sponge function, cube attack

1 Introduction

In 2007, the U.S. National Institute of Standards and Technology (NIST) announced a public contest aiming at the selection of a new standard for a cryptographic hash function. In 2012, after 5 years of intensive competition, the winner was selected. The new SHA-3 standard will be Keccak hash function [8]. In fact, Keccak is a family of sponge functions [7] and it can be used not only as a hash function, but also for generating an infinite bit stream, making it suitable to work as a stream cipher or a pseudorandom bit generator.

In this paper we present attacks on scaled-down Keccak variants, in which the number of rounds is reduced from the full 24. We use the cube attack — an algebraic technique which exploits a low degree polynomial representation of a given cryptographic primitive. This attack is particularly efficient against round-reduced Keccak variants, as the degree of a single Keccak round is only 2. This was already demonstrated by [13], where the cube attack was applied with low complexity to 4-round Keccak. Furthermore, the low algebraic degree of a Keccak round was exploited in more theoretical attacks (with a much higher complexity) such as [1, 6]. In this paper, however, we are interested in practical results, and describe cube attacks and cube testers (which are extensions of the cube attack) on up to 6-round Keccak (and a slightly stronger version containing 6.5 rounds), thus reaching two more rounds than [13].

All the presented attacks have practical complexities and were verified on a desktop PC. Table 1 shows the state-of-the-art regarding practical complexity attacks on Keccak sponge function.

Table 1: Best known practical attacks on the Keccak sponge function. All the reported attacks are on the 1600-bit state variant.

Rounds	Mode	Type of attack	Reference
2	hash function	preimage	[11, 14]
4	hash function	collision	[9]
4	MAC	key recovery	[13]
5	MAC	key recovery	Section 4.1
6	stream cipher	key recovery	Section 4.2

The paper is organized as follows. First, we present a brief description of the cube attack highlighting the key ideas of this technique. Then, a description of the Keccak Sponge Function is given. Next, in Section 4, our cube attack on Keccak is presented. Finally, Sections 5 shows how to use a cube tester to detect a non-random behavior for the 6.5-round variant of the Keccak permutation.

2 Cube Attacks

The cube attack is a chosen plaintext key-recovery attack, which was formally introduced in [10] as an extension of higher order differential cryptanalysis [12] and AIDA [15]. Since its introduction, the cube attack was applied to many different cryptographic primitives such as [2, 4, 13]. Below we give a brief description of the cube attack, and refer the reader to [10] for more details.

The cube attack assumes that the output bit of a cipher is given as a black-box polynomial $f : X^n \to \{0, 1\}$ in the input bits (variables). The main observation used in the attack is that when this polynomial has a (low) algebraic degree d, then summing its output over 2^{d-1} inputs in which a subset of variables (i.e., a cube) of size $d - 1$ ranges over all possible values, and the other variables are fixed to some constant, yields a linear function (see the theorem below).

Theorem 1. *(Dinur, Shamir) Given a polynomial $f : X^n \to \{0, 1\}$ of degree d. Suppose that $0 < k < d$ and t is the monomial $x_0 \dots x_{k-1}$. Write the function as*

$$f(x) = t \cdot P_t(x) + Q_t(x)$$

where none of the terms in $Q_t(x)$ is divisible by t. Note that $\deg P_t \leq d - k$. Then the sum of f over all values of the cube (defined by t) is

$$\sum_{x'=(x_0,\dots,x_{k-1})\in C_t} f(x', x) = P_t(\underbrace{1, \dots, 1}_{k}, x_k, \dots, x_{n-1})$$

whose degree is at most $d - k$ (or 1 if $k = d - 1$), where the cube C_t contains all binary vectors of the length k.

In Appendix D we give a simple combinatorial proof of this theorem. Algebraically, we note that addition and subtraction are the same operation over $GF(2)$. Consequently, the cube sum operation can be viewed as differentiating the polynomial with respect to the cube variables, and thus its degree is reduced accordingly.

2.1 Preprocessing (Offline) Phase

The preprocessing phase is carried out once per cryptosystem and is independent of the value of the secret key.

Let us denote public variables (variables controlled by the attacker e.g., a message or a nonce) by $v = (v_1, \dots, v_{d-1})$ and secret key variables by $x = (x_1, \dots, x_n)$. An output (ciphertext bit, keystream bit, or a hash bit) is determined by the polynomial $f(v, x)$, and we denote

$$\sum_{v\in C_t} f(v, x) = L(x)$$

for some cube C_t, where $L(x)$ is called the *superpoly* of C_t. Assuming that the degree of $f(v, x)$ is d, then according to the main observation

$$L(x) = a_1x_1 + \dots + a_nx_n + c$$

In the preprocessing phase we find linear superpolys $L(x)$ which eventually help us build a set of linear equations in the secret variables. We interpolate the linear coefficients of $L(x)$ as follows

– find the constant $c = \sum_{v \in C_t} f(v, 0)$
– find $a_i = \sum_{v \in C_t} f(v, 0, \ldots, \underbrace{1}_{x_i}, 0, \ldots, 0)) = a_i$

We note that in the most general case, the full symbolic description of $f(v, x)$ is unknown and we need to estimate its degree d using an additional complex preprocessing step. This step is carried out by trying cubes of different dimensions, and testing their *superpolys* $L(x)$ for linearity. However, as described in our specific attacks on Keccak, the degree of $f(v, x)$ can be easily estimated in our attacks, and thus this extra step is not required.

2.2 Online Phase

The online phase is carried out after the secret key is set. In this phase, the attack exploits the ability the choose the values of the public variables v: For each cube C_t, the attacker computes the binary value b_t by summing over the cube C_t.

$$\sum_{v \in C_t} f(v, x) = b_t$$

For a given cube C_t, b_t is equal to the linear expression $L(x)$ determined in the preprocessing phase, therefore a single linear equation is obtained

$$a_1 x_1 + \ldots + a_n x_n + c = b_t.$$

Considering many different cubes C_t, the attacker aims at constructing a sufficient number of linear equations. If the number of (linearly independent) equations is equal to a number of secret variables, the system is solved by Gaussian elimination.[5]

2.3 Cube Testers

The notion of cube testers was introduced in [2], as an extension of the cube attack. Unlike standard cube attacks, cube testers aim at detecting non-random behaviour (rather than performing key recovery), e.g., by observing that the cube sums are always equal to zero, regardless of the value of the secret key. We note that zero-sum distinguishers (proposed in [1], and applied to the Keccak permutation in several other papers) are closely related to cube testers, but they assume that the attacker has the power to choose the public variables at the middle of the permutation (rather than only at the beginning, as in standard attack models).

3 Keccak Sponge Function

Keccak is a family of sponge functions [7]. It can be used as a hash function, but can also generate an infinite bit stream, making it suitable to work as a stream cipher or a pseudorandom bit generator. In this section, we provide a brief description of the Keccak sponge function to the extent necessary for understanding the attacks described in the paper. For a complete specification, we refer the interested reader to the original specification [8].

A sponge function works on an internal state, divided according to two main parameters r and c, which are called bitrate and capacity, respectively. Initially, the $(r + c)$-bit state is filled with 0's, and the message is split into r-bit blocks. Then, the sponge function processes the message in two phases.

[5] More generally, one can use any number of linearly independent equations in order to speed up exhaustive search for the key.

In the first phase (also called the absorbing phase), the r-bit message blocks are XORed into the state, interleaved with applications of the internal permutation. After all message blocks have been processed, the sponge function moves to the second phase (also called the squeezing phase). In this phase, the first r bits of the state are returned as part of the output, interleaved with applications of the internal permutation. The squeezing phase is finished after the desired length of the output digest has been produced.

Keccak is a family of sponge functions defined in [8]. The state of Keccak can be visualized as an array of 5×5 lanes, where each lane is a 64-bit string in the default version (and thus the default state size is 1600 bits). Other versions of Keccak are defined with smaller lanes, and thus smaller state sizes (e.g., a 400-bit state with a 16-bit lane). The state size also determines the number of rounds of the Keccak-f internal permutation, which is 24 for the default 1600-bit version.

All Keccak rounds are the same except for round-dependant constants which are XORed into the state. Below there is a pseudo-code of a single round. In the latter part of the paper, we often refer to the algorithm steps (denoted by Greek letters) described in the following pseudo-code.

```
Round(A,RC) {

θ step
C[x] = A[x,0] xor A[x,1] xor A[x,2] xor
       A[x,3] xor A[x,4],                                        forall x in (0...4)
D[x] = C[x-1] xor rot(C[x+1],1),                                 forall x in (0...4)
A[x,y] = A[x,y] xor D[x],                              forall (x,y) in (0...4,0...4)

ρ step                                                 forall (x,y) in (0...4,0...4)
A[x,y] = rot(A[x,y], r[x,y]),

π step                                                 forall (x,y) in (0...4,0...4)
B[y,2*x+3*y] = A[x,y],

χ step                                                 forall (x,y) in (0...4,0...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   }
```

All the operations on the indices shown in the pseudo-code are done modulo 5. `A` denotes the complete permutation state array and `A[x,y]` denotes a particular lane in that state. `B[x,y]`, `C[x]`, `D[x]` are intermediate variables. The constants `r[x,y]` are the rotation offsets, while `RC` are the round constants. `rot(W,m)` is the usual bitwise rotation operation, moving bit at position i into position $i+m$ in the lane `W` ($i+m$ are done modulo the lane size). θ is a linear operation that provides diffusion to the state. ρ is a permutation that mixes bits of a lane using rotation and π permutes lanes. The only non-linear operation is χ, which can be viewed as a layer of 5-bit S-boxes. Note that the algebraic degree of χ over $GF(2)$ is only 2. Finally, ι XORes the round constant into the first lane.

In this paper we refer to the linear steps θ, ρ, π as the first half of a round, and the remaining steps χ and ι as the second half of a round.

Keccak Working as MAC The Keccak sponge function can be used in keyed mode providing many different functionalities. One of these functionalities is a hash-based message authentication code (MAC), which is used for verifying the data integrity and the authentication of a message. A hash-based algorithm for calculating a MAC involves a cryptographic hash function in combination with a secret key. A typical construction is HMAC proposed by Bellare et al. [5]. However, for the Keccak hash function, the nested approach of HMAC is not needed and in order to provide a MAC functionality, we simply prepend the secret key to the message.

4 Cube Attack on Keccak Sponge Function

We describe key-recovery attacks on round-reduced Keccak, when it is used as a MAC and in the stream cipher mode.

4.1 Attack on 5-round Keccak Working as MAC

We attack the default variant of Keccak with a 1600-bit state ($r = 1024$, $c = 576$), where the number of rounds is reduced to 5. The key size and authentication tag are both 128 bits long — typical values for MAC applications. We assume that the attacker can calculate any MAC for a chosen message, and aims to recover the 128-bit secret key.

In this attack, the 1600-bit state consists of 128 secret variables (the secret key) and 128 public variables (message variables). The remaining bits are set to '0', except for the padding. The attacker collects MACs for chosen 128-bit messages and then deduces the secret key from the collected message-MAC pairs.

As previously noted, we exploit the property that the algebraic degree of a single round of the Keccak permutation is only 2. Therefore, after 5 rounds the algebraic degree is at most $2^5 = 32$.

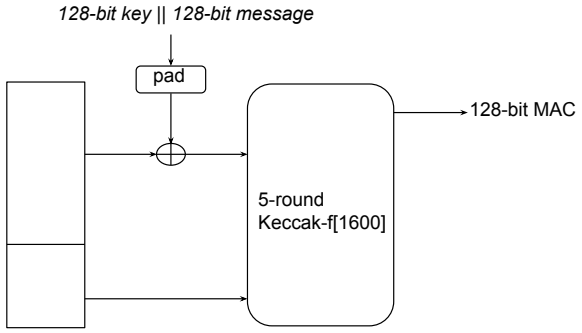

Fig. 1: Settings for the key-recovery attack on 5-round Keccak generating a MAC.

Preprocessing Phase As the algebraic degree of 5 Keccak rounds is at most 32, then for any cube of 31 variables, the superpoly only consists of linear terms. Thus, we can avoid the step of testing the superpolys for linearity. On the other hand, the superpolys can be constants, which are not useful for key-recovery attacks (as they do not contain information about the key). This typically occurs due to the slow diffusion of variables into the initial rounds, which causes the algebraic degree of the examined output bits to be less than the maximal possible degree of 32.

To find useful cubes for out attack, we randomly picked 31 out of the 128 public variables and checked whether the superpoly consists of any secret variables or it is constant. With this simple

strategy, we were able to find 117 linearly independent expressions (superpolys) in a few days on a desktop PC. The search was somewhat more complex than expected, as we observed that only $20-25\%$ of the superpolys were useful (i.e., non-constant). On the other hand, we observed that in many cases, for a given cube, if we examine different outputs (with their corresponding superpolys), we can find many superpolys and shorten the search time. In Appendix A we give the cubes chosen for the attack.

Online Phase

In the online phase, the attacker computes the actual binary value of a given superpoly by summing over the outputs obtained from the corresponding cube. There are 19 cubes used in this attack, each cube with 31 variables. Thus, the attacker obtains $19 \cdot 2^{31} \cong 2^{35}$ outputs for 5-round Keccak. Having computed the values of the superpolys, the attacker constructs a set of 117 linearly independent equations, and recovers the full 128-bit secret key by guessing the values of 11 additional linearly independent equations. In total, the complexity of the online phase is dominated by 2^{35} Keccak calls (the linear algebra complexity and the additional 2^{11} guesses can be neglected).

4.2 Attack on 6-round Keccak Working in Stream Cipher Mode

A direct extension of the attack to 6 rounds seems infeasible as we would deal with polynomials of approximate degree $2^6 = 64$ and it is very unlikely to find (in reasonable time) cubes with linear superpolys. However, one more round can be reached by exploiting an additional property of Keccak: as χ operates on the rows independently, if a whole row (5 bits) is known, we can invert these bits through ι and χ from the originally given output bits.

On the other hand, 128 output bits (a generated MAC) are not sufficient for inversion — these bits do not allow us to uniquely calculate any bit at the input. Considering longer MACs, e.g., 320-bit MACs, makes the attack setting somewhat artificial. However, we can still attack the Keccak sponge function working in a different mode, where the attacker has access to more output bits, such as the stream cipher mode.

We now describe a reasonable attack setting on 6-round Keccak. We attack the default variant of Keccak with 1600-bit state, $r = 1024$, $c = 576$. The state is initialized with a 128-bit key concatenated with a 128-bit public Initialization Vector (IV). After a single Keccak permutation call, a 1024-bit keystream is extracted and used to encrypt a plaintext via bitwise XOR (as shown in Figure 2).

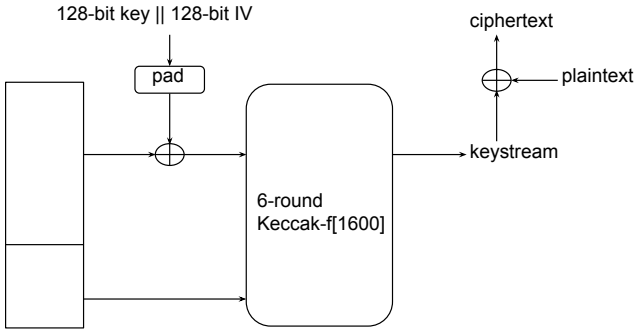

Fig. 2: 6-round Keccak used in the stream cipher mode

As we assume that the attacker can obtain the keystreams for various choices of IVs, it is a similar scenario to the previous attack on the 5-round Keccak MAC. On the other hand, the attacker now

possesses 1024 output bits (keystream bits) and is able to invert them through ι and χ. Consequently, the final nonlinear step χ can be omitted and the cube attack is reduced to 5.5 rounds, for which the output bits have a manageable polynomial degree of at most 32 (The first half a round is linear and does not increase the polynomial degree, hence 5.5 rounds in total.)

We executed the preprocessing phase in a similar way to the one described for the 5-round attack. We were able to find 128 linearly independent superpolys using 25 cubes (listed in Appendix B). This gives an online attack complexity of $2^{31} \cdot 25 \cong 2^{36}$.

4.3 Attack on State-reduced 6-round Keccak Working as MAC

The Keccak sponge function can also work on smaller states which may be useful for lightweight cryptography. We attacked the Keccak MAC operating on a 400-bit state with an 80-bit key. As the state is smaller, 128 bits of MAC (output bits) cover the complete rows in the state and we are able to invert these rows through ι and χ. Therefore, we could attack the 6-round Keccak MAC in practical time.

During the preprocessing phase, we found 80 linearly independent superpolys using 18 cubes. This allows to recover the 80-bit secret key with complexity $2^{31} \cdot 18 \cong 2^{35}$. It is interesting to note that, compared to the previous attacks, the superpolys consist of many more secret variables. It is due to a faster diffusion of variables into the smaller state. Examples of the cubes chosen for this attack are given in Appendix C.

5 Practical Complexity Cube Tester for 6.5-round Keccak Permutation

In this section we show how to construct a practical cube tester for the 6.5-round Keccak permutation. As the expected algebraic degree for 6-round Keccak is 64, such an attack may seem at first impractical to mount on a desktop PC (without exploiting some internal invertibility properties, as in the previous section). However, if we carefully choose the initial state and exploit a special property of θ, we can considerably reduce the output degree after 6 rounds and keep the complexity practical.

The exploited (well-known) property of θ is that its action depends only on the column parities (and not on the actual values of the 5 bits inside each column). Thus, if we place the cube variables in such a way that all the column parities are constants for all values of the cube variables, then θ will not diffuse these variables throughout the state. Moreover, as ρ and π only permute the bits in the state, it is easy to choose the cube variables such that after the linear part of the round, they do not interact with each other through the subsequent non-linear χ layer. Consequently, the algebraic degree of the polynomial in the cube variables remains 1 after the first round, and it is at most 32 after 6 rounds.

Thus, we choose a cube of 33 variables (as shown in Figure 3), while the remaining bits of the input state are set to arbitrary constants (some of which are potentially unknown secret variables). Since the degree of the output polynomials in the cube variables after 6 rounds is only 32, the cube sum of any output bit after 6 rounds is equal to zero, which is a clear non-random property. Moreover, we can add a (linear) half of a round and obtain a 6.5-round distinguisher using the same cube. Furthermore, if we assume that we can obtain sufficiently many output bits in order to partially invert the non-linear layer (as in the previous section), we can extend the attack to 7 rounds in practical time. Note that the distinguishing attack works regardless of the number of secret variables, their location, or their values.

Finally, we note that a direct application of this cube tester for a key recovery is not straightforward, and (unlike the previous attacks) requires testing superpolys for linearity (in fact, linear superpolys are not guaranteed to exist).

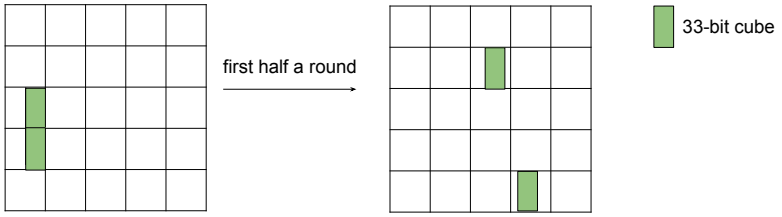

Fig. 3: The initial state of a cube tester and the transition through the first linear part of the round (θ, ρ, π steps)

6 Conclusion

We mounted the cube attack on the Keccak sponge function and showed that 5- and 6-round variants are vulnerable to this technique. In particular, we attacked the Keccak MAC and the variant working in the stream cipher mode. We also reported results on a state-reduced variant of Keccak, and since the full-round version of this variant has fewer than 24 rounds, it also provides a (slightly) smaller security margin against our attacks. Finally, we described the 6.5-round cube tester — a testable way to exhibit a non-random behaviour of the permutation.

References

1. Aumasson, J.P., Meier, W.: Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi. Tech. rep., NIST mailing list (2009)
2. Aumasson, J.P., Dinur, I., Meier, W., Shamir, A.: Cube testers and key recovery attacks on reduced-round md6 and trivium. In: FSE. pp. 1–22 (2009)
3. Aumasson, J.P., Khovratovich, D.: First Analysis of Keccak, `http://131002.net/data/papers/AK09.pdf`
4. Bard, G.V., Courtois, N., Nakahara, J., Sepehrdad, P., Zhang, B.: Algebraic, AIDA/Cube and Side Channel Analysis of KATAN Family of Block Ciphers. In: INDOCRYPT. pp. 176–196 (2010)
5. Bellare, M., Canetti, R., Krawczyk, H.: Message authentication using hash functions: the HMAC construction. CryptoBytes 2(1), 12–15 (1996)
6. Bernstein, D.J.: Second preimages for 6 (7? (8??)) rounds of Keccak? NIST mailing list (2010), `http://ehash.iaik.tugraz.at/uploads/6/65/NIST-mailing-list_Bernstein-Daemen.txt`
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic sponges, `http://sponge.noekeon.org/CSF-0.1.pdf`
8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak sponge function family main document, `http://keccak.noekeon.org/Keccak-main-2.1.pdf`
9. Dinur, I., Dunkelman, O., Shamir, A.: New Attacks on Keccak-224 and Keccak-256. In: Canteaut, A. (ed.) Fast Software Encryption, Lecture Notes in Computer Science, vol. 7549, pp. 442–461. Springer Berlin Heidelberg (2012)
10. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. In: EUROCRYPT. pp. 278–299 (2009)
11. Homsirikamol, E., Morawiecki, P., Rogawski, M., Srebrny, M.: Security margin evaluation of SHA-3 contest finalists through SAT-based attacks. In: 11th Int. Conf. on Information Systems and Industrial Management. LNCS, vol. 7564. Springer Berlin Heidelberg (2012)
12. Lai, X.: Higher order derivatives and differential cryptanalysis. In: Blahut, R., Costello, DanielJ., J., Maurer, U., Mittelholzer, T. (eds.) Communications and Cryptography, The Springer International Series in Engineering and Computer Science, vol. 276, pp. 227–233. Springer US (1994)
13. Lathrop, J.: Cube Attacks on Cryptographic Hash Functions. Master's thesis, Rochester Institute of Technology (2009)
14. Naya-Plasencia, M., Röck, A., Meier, W.: Practical analysis of reduced-round keccak. In: Bernstein, D., Chatterjee, S. (eds.) Progress in Cryptology INDOCRYPT 2011, Lecture Notes in Computer Science, vol. 7107, pp. 236–254. Springer Berlin Heidelberg (2011)
15. Vielhaber, M.: Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack. Cryptology ePrint Archive, Report 2007/413 (2007)

Appendix

A

Table 2: Cubes and corresponding superpolys used in the attack on 5-round Keccak MAC.

cube: 128,130,131,139,145,146,147,148,151,155,158,160,161,163,164,165,185,186,189,190,193,196,205,212,220,225,229,238,242,245,249			
superpoly	output bit	superpoly	output bit
x_{77}	7	$1 + x_{110}$	13
$1 + x_{113}$	15	x_{25}	31
$1 + x_{103}$	42	$1 + x_{105}$	69
x_{44}	84	x_{123}	87
$1 + x_{100}$	96	$1 + x_{104}$	100
x_{17}	112	$x_{38} + x_{51}$	71
$1 + x_7 + x_{19}$	91	$1 + x_{80} + x_{122}$	113
$x_{17} + x_{68} + x_{116}$	114		
cube: 128,129,134,135,138,139,141,151,154,155,157,166,168,171,175,180,191,193,198,202,203,206,209,214,216,222,223,225,239,246,247			
superpoly	output bit	superpoly	output bit
$1 + x_{50}$	7	$1 + x_{124}$	16
$1 + x_{109}$	19	x_{92}	20
$1 + x_{101}$	30	$1 + x_{85}$	35
x_{15}	37	$1 + x_{67}$	53
$1 + x_4$	55	$1 + x_{90}$	76
$1 + x_{126}$	94	$1 + x_{84}$	95
$1 + x_{33}$	108	x_6	112
$1 + x_{78}$	122	$1 + x_{48}$	124
$x_{39} + x_{99}$	27	$x_{68} + x_{117}$	115
$1 + x_{23} + x_{87}$	119	$1 + x_{27} + x_{50} + x_{81}$	121
cube: 129,134,135,139,140,146,159,166,170,171,174,176,179,180,182,184,188,192,193,200,202,207,208,212,214,219,225,234,243,248,252			
superpoly	output bit		
x_{40}	100		
cube: 128,132,134,136,137,139,140,155,158,159,161,162,166,168,175,177,179,183,185,189,195,209,210,211,214,234,239,247,249,251,253			
superpoly	output bit	superpoly	output bit
x_{13}	9	x_{72}	22
cube: 128,136,144,145,152,154,164,170,171,173,182,187,190,192,194,198,200,201,205,212,215,217,220,227,230,234,240,242,245,249,253			
superpoly	output bit	superpoly	output bit
x_{76}	90	x_{39}	113
cube: 133,134,138,139,140,141,152,153,156,157,158,163,166,170,171,173,189,194,207,215,216,221,231,234,235,238,241,250,253,254,255			
superpoly	output bit	superpoly	output bit
$1 + x_{119}$	115	$1 + x_{35}$	121
cube: 130,135,139,140,143,147,150,165,169,172,179,181,182,186,193,197,205,209,219,226,228,230,231,233,234,238,239,240,245,246,252			
superpoly	output bit		
x_{107}	25		
cube: 132,137,139,140,143,147,152,158,161,163,165,169,171,182,184,186,187,190,192,198,199,200,223,227,229,233,239,242,244,245,246			
superpoly	output bit	superpoly	output bit
$1 + x_{88}$	3	$1 + x_{49} + x_{94}$	19
cube: 135,136,141,145,150,156,158,160,161,165,169,170,172,177,186,189,190,197,201,204,211,212,213,226,228,240,241,242,250,251,254			
superpoly	output bit	superpoly	output bit
$1 + x_{111}$	10	x_{66}	42
x_5	118		
cube: 130,135,138,140,141,146,153,155,165,170,178,199,200,206,209,214,218,222,223,224,226,228,231,232,233,238,243,251,252,253,255			
superpoly	output bit	superpoly	output bit
x_{91}	16	$1 + x_{34}$	99
cube: 128,129,130,132,137,142,152,153,155,160,164,165,166,175,187,196,200,201,205,212,217,221,222,226,228,235,237,242,243,246,252			
superpoly	output bit	superpoly	output bit
x_{96}	16	$1 + x_{57}$	19
$1 + x_{70}$	94	x_{53}	96
x_{30}	113	$x_3 + x_{43}$	22
$1 + x_{42} + x_{106}$	30	$x_{63} + x_{67}$	91

$x_{26} + x_{49} + x_{113}$	76		
cube: 128,129,135,137,140,145,150,152,162,163,164,166,170,175,179,181,186,187,198,202,209,216,220,221,222,230,234,240,241,245,248			
superpoly	output bit	superpoly	output bit
$1 + x_{46}$	0	$1 + x_{83}$	8
x_{118}	19	x_{64}	27
x_{49}	32	$1 + x_{3}$	51
x_{58}	63	$1 + x_{114}$	66
$1 + x_{24}$	74	x_{38}	82
$1 + x_{65}$	90	$1 + x_{125}$	92
$1 + x_{16}$	102	x_{87}	119
x_{71}	121	$x_{0} + x_{24}$	31
$x_{1} + x_{72}$	34	$x_{32} + x_{45} + x_{66} + x_{104}$	23
cube: 129,130,136,142,149,153,156,157,159,165,166,173,175,181,188,190,193,194,195,205,209,211,219,221,225,234,239,245,247,253,254			
superpoly	output bit	superpoly	output bit
x_{80}	7	$1 + x_{47}$	12
x_{8}	20	$1 + x_{97}$	72
$1 + x_{86}$	120	$x_{59} + x_{119}$	32
$1 + x_{62} + x_{107}$	76	$x_{29} + x_{42}$	111
cube: 131,132,134,136,138,142,143,147,152,158,165,167,171,172,173,180,186,196,206,208,213,214,217,219,226,233,235,237,239,250,251			
superpoly	output bit	superpoly	output bit
x_{74}	5	x_{102}	16
$1 + x_{10}$	47	x_{19}	49
$1 + x_{11}$	57	x_{69}	69
$1 + x_{127}$	79	x_{2}	101
x_{112}	116		
cube: 135,138,142,149,151,153,156,162,163,165,166,173,174,178,182,187,188,192,193,210,214,218,219,221,228,235,237,238,243,252,255			
superpoly	output bit	superpoly	output bit
x_{81}	30	$1 + x_{75}$	121
$x_{79} + x_{98}$	22		
cube: 136,137,141,145,148,152,155,157,162,166,169,184,186,188,189,203,204,209,210,214,217,221,223,225,227,229,237,243,247,248,252			
superpoly	output bit	superpoly	output bit
$1 + x_{52}$	61	$1 + x_{95}$	79
$1 + x_{115}$	93	$1 + x_{60}$	98
x_{117}	117	$1 + x_{14} + x_{78}$	44
$x_{28} + x_{66}$	62	$1 + x_{37} + x_{112}$	72
$1 + x_{41} + x_{104}$	106	$x_{85} + x_{108}$	113
cube: 128,131,133,134,145,146,147,150,151,154,161,163,164,168,174,180,181,190,198,203,205,206,209,217,224,225,232,245,247,248,255			
superpoly	output bit	superpoly	output bit
$1 + x_{18}$	24	$1 + x_{82}$	25
x_{73}	53	x_{42}	77
$x_{48} + x_{61}$	89		
cube: 130,131,133,134,135,145,148,151,153,164,165,179,180,189,191,193,194,199,207,214,221,222,224,225,229,231,237,239,246,253,255			
superpoly	output bit	superpoly	output bit
$1 + x_{93}$	11	$1 + x_{89}$	47
$1 + x_{32} + x_{95} + x_{100}$	31		
cube: 138,141,156,157,164,166,180,184,185,188,189,194,196,198,201,203,204,212,214,215,216,220,222,230,232,238,240,247,248,249,251			
superpoly	output bit	superpoly	output bit
$1 + x_{12}$	23	$1 + x_{56}$	119

B

Table 4: Cubes and corresponding superpolys found for 5.5 rounds, used in the attack on the 6-round Keccak working in the stream cipher mode.

cube: 128,133,134,137,138,145,153,154,155,157,158,161,175,180,182,187,191,192,195,199,206,208,211,220,227,229,245,247,249,251,252			
superpoly	output bit	superpoly	output bit
x_{76}	1	$1 + x_{64}$	13
x_{41}	17	x_{106}	28
$1 + x_{85}$	38	$1 + x_{32}$	46

$1 + x_{10}$	49	x_0	70
x_{109}	71	$1 + x_{121}$	73
$1 + x_{25}$	88	x_{96}	91
$1 + x_{35}$	95	$1 + x_{68}$	97
x_{42}	106	x_{72}	111
x_{26}	112	$1 + x_{34}$	123
x_{116}	125		
cube: 128,132,134,135,139,144,149,154,155,156,157,159,168,171,181,184,191,195,201,211,217,225,226,229,231,232,234,239,240,247,248			
superpoly	output bit	superpoly	output bit
x_{67}	18	$1 + x_{81}$	20
$1 + x_{97}$	23	$1 + x_{87}$	42
$1 + x_{66}$	43	$1 + x_{108}$	48
$1 + x_{80}$	61	$1 + x_{88}$	103
$1 + x_{95}$	111	x_{78}	114
x_{91}	119		
cube: 131,136,137,138,141,143,154,156,160,164,170,173,178,180,183,185,188,193,201,202,217,221,225,231,239,243,244,245,249,250,251			
superpoly	output bit	superpoly	output bit
x_{92}	19	$1 + x_{105}$	24
x_{33}	45	x_{101}	54
x_{89}	74	$1 + x_{126}$	107
cube: 128,130,138,143,144,145,147,159,162,163,169,170,177,178,179,180,184,185,189,195,200,207,208,216,220,221,222,242,243,252,255			
superpoly	output bit	superpoly	output bit
x_{82}	1	x_{48}	2
x_{45}	14	$1 + x_{37}$	20
x_{110}	48	$1 + x_{13}$	68
$1 + x_{83}$	88	$1 + x_{127}$	110
cube: 136,138,139,141,142,144,152,154,158,162,163,169,177,187,189,195,196,201,207,209,214,218,221,224,228,234,236,239,240,243,244			
superpoly	output bit	superpoly	output bit
$1 + x_{65}$	5	x_{86}	6
$1 + x_4$	9	$1 + x_{70}$	10
x_{54}	12	$1 + x_{22}$	14
$1 + x_{58}$	18	$1 + x_{55}$	27
x_{104}	34	x_{40}	35
$1 + x_{39}$	36	x_{28}	40
x_{53}	53	$1 + x_{21}$	68
x_{99}	69	x_1	89
x_{100}	95	x_{20}	115
x_{52}	116	x_{31}	121
$x_{23} + x_{65}$	11		
cube: 128,131,132,137,142,145,147,148,154,155,163,174,176,180,190,192,195,197,199,207,215,217,220,228,232,233,235,238,239,241,247			
superpoly	output bit	superpoly	output bit
x_{57}	0	$1 + x_{38}$	13
x_7	15	$1 + x_{117}$	33
x_{69}	50	$1 + x_9$	62
x_{77}	72	x_{46}	74
$1 + x_{30}$	82	$1 + x_{75}$	89
$1 + x_{60}$	118		
cube: 128,131,136,138,142,143,148,151,153,155,158,159,160,163,177,180,186,187,189,192,193,195,198,223,229,231,236,247,248,254,255			
superpoly	output bit	superpoly	output bit
$1 + x_{98}$	9	x_{122}	37
x_{102}	52	$1 + x_{114}$	58
$1 + x_{79}$	61	$1 + x_{113}$	65
x_{118}	68	x_{24}	74
x_{61}	77	x_{43}	104
cube: 129,138,144,161,162,163,164,170,171,176,183,188,190,193,197,200,205,207,212,214,215,216,219,226,227,231,233,239,247,251,252			
superpoly	output bit		
x_{29}	116		
cube: 132,133,140,145,153,157,163,164,165,168,170,171,175,197,198,204,209,210,214,220,222,227,228,233,234,237,239,240,244,247,249			
superpoly	output bit	superpoly	output bit
x_{62}	10	$1 + x_{71}$	17

x_3	69	$1 + x_{84}$	78
x_{123}	82	x_{63}	117
cube: 137,138,147,149,155,156,157,163,168,171,183,192,194,195,197,201,202,204,205,208,211,216,217,218,220,226,227,228,229,247,252			
superpoly	output bit		
x_{19}	102		
cube: 128,129,131,134,137,141,142,146,148,152,161,171,175,178,180,192,193,198,200,202,203,207,208,209,216,218,223,224,236,243,255			
superpoly	output bit	superpoly	output bit
x_6	46	x_8	61
x_{17}	103	$1 + x_{124}$	126
cube: 128,132,140,142,145,148,152,157,162,164,168,175,177,185,187,190,194,195,196,199,200,203,204,214,219,223,231,237,239,248,250			
superpoly	output bit	superpoly	output bit
x_{103}	24	$1 + x_{94}$	30
$1 + x_{56}$	75		
cube: 140,141,144,146,151,153,154,157,158,162,165,168,170,176,202,204,206,219,220,226,227,230,232,234,238,241,242,243,244,250,251			
superpoly	output bit		
$1 + x_{12}$	104		
cube: 128,135,136,137,146,150,168,171,178,180,184,189,193,197,198,207,212,214,215,217,218,219,220,227,230,233,236,240,241,246,247			
superpoly	output bit	superpoly	output bit
x_{125}	62	$1 + x_{27}$	89
cube: 129,132,146,151,152,160,161,165,169,182,184,187,197,198,203,204,211,215,219,228,229,231,233,239,240,246,247,248,249,253,255			
superpoly	output bit		
$1 + x_{120}$	127		
cube: 129,131,134,138,139,141,147,149,155,170,175,181,185,186,198,200,202,204,207,208,217,223,235,236,240,241,242,244,246,247,253			
superpoly	output bit	superpoly	output bit
$1 + x_{47}$	2	$1 + x_{15}$	5
$1 + x_{93}$	17	x_{16}	19
x_{44}	65	$1 + x_{115}$	124
x_{59}	127		
cube: 135,138,141,143,146,149,155,156,160,167,177,181,184,190,191,194,201,210,220,223,227,234,235,237,242,244,246,249,250,253,254			
superpoly	output bit	superpoly	output bit
$1 + x_{90}$	11	x_{14}	44
$1 + x_{119}$	67	$1 + x_{111}$	114
cube: 132,133,134,138,139,143,150,153,154,159,170,173,178,181,183,194,198,200,207,211,215,228,230,237,238,240,241,245,249,250,254			
superpoly	output bit	superpoly	output bit
x_{36}	67	x_{74}	82
cube: 130,134,137,138,150,155,161,164,165,168,169,172,173,174,177,178,183,184,187,192,199,212,213,215,220,223,225,230,232,235,236			
superpoly	output bit	superpoly	output bit
$1 + x_2$	101	x_{112}	112
cube: 130,132,138,142,152,157,158,159,167,170,175,177,181,183,185,186,192,199,203,206,207,218,220,231,236,239,244,245,246,247,254			
superpoly	output bit		
x_{18}	44		
cube: 134,137,138,140,143,144,146,149,152,155,162,164,180,183,185,186,196,202,207,217,221,224,235,237,239,246,248,249,251,253,254			
superpoly	output bit	superpoly	output bit
x_{50}	7	x_5	102
x_{73}	111		
cube: 130,131,132,134,136,140,143,148,149,151,165,166,174,182,184,190,192,195,204,205,220,221,223,226,228,234,241,242,243,245,251			
superpoly	output bit		
$1 + x_{51}$	21		
cube: 134,142,143,146,147,152,160,161,170,172,173,175,181,182,183,191,200,202,205,207,213,220,221,223,226,228,232,237,238,243,248			
superpoly	output bit		
$1 + x_{11}$	82		
cube: 141,143,146,155,156,159,162,167,171,172,173,174,177,179,180,182,184,199,203,204,206,207,208,209,219,222,228,241,242,250,253			
superpoly	output bit		
$1 + x_{107}$	50		
cube: 131,132,133,140,149,155,156,160,163,168,174,178,182,185,186,195,196,204,213,215,216,224,225,231,232,234,236,237,245,252,253			
superpoly	output bit		
$1 + x_{49}$	61		

C

Table 6: Example cubes and corresponding superpolys found for the 5.5-round variant with the reduced (400-bit) state. Cubes were used in the attack on the 6-round Keccak MAC.

cube: 80,82,84,85,87,90,91,96,102,105,109,110,111,116,119,122,128,130,133,134,136,139,140,141,145,146,147,149,153,156,159			
superpoly	output bit	superpoly	output bit
$1 + x_1 + x_2 + x_8 + x_{11} + x_{12} + x_{16} + x_{17} + x_{18} + x_{19} + x_{20} + x_{31} + x_{35} + x_{37} + x_{40} + x_{41} + x_{50} + x_{52} + x_{62} + x_{65} + x_{69} + x_{71} + x_{74} + x_{79}$	29	$x_2 + x_4 + x_5 + x_{16} + x_{17} + x_{20} + x_{22} + x_{24} + x_{28} + x_{34} + x_{40} + x_{42} + x_{43} + x_{44} + x_{47} + x_{49} + x_{51} + x_{52} + x_{53} + x_{54} + x_{56} + x_{60} + x_{61} + x_{62} + x_{67} + x_{69} + x_{72} + x_{73} + x_{75} + x_{78}$	98
$x_0 + x_2 + x_4 + x_7 + x_8 + x_{10} + x_{11} + x_{13} + x_{14} + x_{16} + x_{17} + x_{20} + x_{23} + x_{26} + x_{28} + x_{30} + x_{31} + x_{32} + x_{34} + x_{35} + x_{36} + x_{39} + x_{41} + x_{43} + x_{46} + x_{49} + x_{52} + x_{54} + x_{56} + x_{63} + x_{76}$	79		
cube: 80,82,89,90,91,92,98,99,103,110,111,118,121,122,123,125,126,128,130,132,133,135,136,137,140,145,148,154,155,157,159			
superpoly	output bit	superpoly	output bit
$x_6 + x_8 + x_9 + x_{10} + x_{11} + x_{15} + x_{17} + x_{21} + x_{29} + x_{34} + x_{35} + x_{43} + x_{47} + x_{48} + x_{50} + x_{56} + x_{57} + x_{58} + x_{59} + x_{60} + x_{67} + x_{68} + x_{69} + x_{72} + x_{77}$	45	$x_0 + x_1 + x_3 + x_9 + x_{12} + x_{13} + x_{14} + x_{15} + x_{16} + x_{22} + x_{26} + x_{30} + x_{31} + x_{34} + x_{36} + x_{41} + x_{42} + x_{45} + x_{47} + x_{49} + x_{53} + x_{61} + x_{64} + x_{67} + x_{69} + x_{73} + x_{74} + x_{77}$	42
$x_0 + x_1 + x_2 + x_3 + x_4 + x_8 + x_{10} + x_{11} + x_{14} + x_{16} + x_{20} + x_{22} + x_{24} + x_{26} + x_{33} + x_{34} + x_{35} + x_{37} + x_{39} + x_{44} + x_{52} + x_{53} + x_{56} + x_{61} + x_{62} + x_{63} + x_{64} + x_{70} + x_{72} + x_{73} + x_{78}$	12		

D

Proof. We can write the values of the polynomial as follows: for all the values of the cube C_t

$$\begin{aligned}
f(0,\ldots,0,x_k,\ldots,x_{n-1}) &= 0 \cdot P_t(0,\ldots,0,x_k,\ldots,x_{n-1}) + Q_t(0,\ldots,0,x_k,\ldots,x_{n-1}) \\
f(0,\ldots,1,x_k,\ldots,x_{n-1}) &= 0 \cdot P_t(0,\ldots,1,x_k,\ldots,x_{n-1}) + Q_t(0,\ldots,1,x_k,\ldots,x_{n-1}) \\
&\vdots \\
f(1,\ldots,1,x_k,\ldots,x_{n-1}) &= 1 \cdot P_t(1,\ldots,1,x_k,\ldots,x_{n-1}) + Q_t(1,\ldots,1,x_k,\ldots,x_{n-1}) \\
\hline
\sum_{x'=(x_0,\ldots,x_{k-1})\in C_t} f(x',x) &= P_t(1,\ldots,1,x_k,\ldots,x_{n-1}) + 0(?)
\end{aligned}$$

As shown above if we sum up all the terms with P_t then there is only case when product of the bits of the cube is equal to 1. Now we need to show that the sum of Q_t over the cube is zero. Q_t itself is a sum of monomials. Each monomial may

- have all the variables different from $x_0,\ldots,x_{k-1}$. This monomial is sum up 2^k times (even number) so its contribution is zero,
- overlap on ℓ-bits of the cube ($\ell < k$). This means that the monomial contains a ℓ variables of the cube. This monomial becomes zero when at least one variable is zero (there are $2^k - 2^{k-\ell}$ such vectors). If all variables are one, then it gets added up $2^{k-\ell}$ times and as this is even number (or $k - \ell \geq 1$), it is equal to zero.

This completes the proof. □