Cube Attacks and Cube-attack-like Cryptanalysis on the Round-reduced Keccak Sponge Function

Abstract. In this paper, we comprehensively study the resistance of keyed variants of SHA-3 (Keccak) against algebraic attacks. This analysis covers a wide range of key recovery, MAC forgery and other types of attacks, breaking up to 9 rounds (out of the full 24) of the Keccak internal permutation much faster than exhaustive search. Moreover, some of our attacks on the 6-round Keccak are completely practical and were verified on a desktop PC. Our methods combine cube attacks (an algebraic key recovery attack) and related algebraic techniques with structural analysis of the Keccak permutation. These techniques should be useful in future cryptanalysis of Keccak and similar designs.
Although our attacks break more rounds than previously published techniques, the security margin of Keccak remains large. For Keyak – a Keccak-based authenticated encryption scheme – the nominal number of rounds is 12 and therefore its security margin is smaller (although still sufficient).

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 scrutiny, the winner was selected. The new SHA-3 standard is the Keccak hash function [8].

As a new standard, Keccak will be widely deployed, and therefore understanding its security is crucial. Indeed, the hash function and its internal permutation have been extensively analysed in many research papers [2, 6, 11–13, 17, 21, 22]. However, most papers focused on key-less variants of the hash function, whereas Keccak is actually a family of sponge functions [7] and can also be used in keyed modes. In such modes, the sponge function can generate an infinite bit stream, making it suitable to work as a stream cipher or as a pseudorandom bit generator. Furthermore, the sponge function can be used as a building block for message authentication codes (MACs) and authenticated encryption (AE) schemes as described in [10].

In this paper, we aim at filling the gap in the security analysis of Keccak by analysing keyed modes of its variants, in which the number of rounds of the internal permutation is reduced from the full 24. We analyse concrete instances of stream ciphers, MACs and AE schemes based on the Keccak permutation and investigate their resistance against strong key recovery attacks and weaker types of attacks (such as a MAC forgery). The stream cipher and MAC constructions we analyse are instantiated according to the design strategy of [10], which describes methods for building such ciphers from the sponge function. The AE scheme we analyse is called Keyak Lane — a recently proposed authenticated encryption scheme, submitted by the Keccak designers to the CAESAR competition [1] for authenticated encryption.

All of our attacks are closely related to high order differential cryptanalysis [20], which used, for the first time, high order derivatives in cryptanalysis of ciphers with low algebraic degree. As the degree of a round of the Keccak internal permutation is only 2, it makes the round-reduced Keccak a natural target for these types of attacks.

Our analysis is divided into three parts. First, we investigate the security of keyed modes of Keccak against cube attacks — an algebraic key recovery technique introduced in [14]. This attack was previously applied to the 4-round Keccak in [21], and we show how to break up to 6 rounds of some Keccak variants. As our optimized cube attacks have very low complexity, they were implemented and verified on a desktop PC.

In the second part of our analysis, we study the security of keyed modes of the round-reduced Keccak against the cube testers [3]. Unlike the cube attack, cube testers do not recover the secret key, but allow to predict outputs of the scheme for previously unseen inputs, giving rise to other types of attacks (such as MAC forgery). Here, we present theoretical attacks on up to 9 rounds of keyed

Keccak variants. Although the attack complexities are impractical, they are significantly faster than generic attacks.

Finally, we reconsider key recovery attacks, and show how to recover the key for 7 rounds of Keccak, much faster than exhaustive search. Table 1 summarizes our attacks on keyed modes of Keccak.

Table 1: Summary of Attacks

Mode	Rounds	Type of Attack	Generic complexity	Attack complexity	Reference
MAC	5	Key Recovery	2^{128}	2^{36}	Sect. 4
MAC	6	Key Recovery	2^{128}	2^{36}	Sect. 6
MAC	7	Key Recovery	2^{128}	2^{97}	Sect. 6
MAC	7	Forgery	2^{128}	2^{65}	Sect. 5
MAC	8	Forgery	2^{256}	2^{129}	Sect. 5
AE (Keyak)	6	Key Recovery	2^{128}	2^{36}	Sect. 4
AE (Keyak)	7	Key Recovery	2^{128}	2^{76}	Sect. 6
AE (Keyak)	7	Forgery	2^{128}	2^{65}	Sect. 5
Stream Cipher	6	Key Recovery	2^{128}	2^{35}	Sect. 4
Stream Cipher	8	Keystream Prediction	2^{256}	2^{128}	Sect. 5
Stream Cipher	9	Keystream Prediction	2^{512}	2^{256}	Sect. 5

The attacks developed in this paper are described with an increasing degree of sophistication. We start by describing rather simple (yet effective) techniques, such as partial inversion of the internal non-linear mapping of Keccak. Then, our more complex methods exploit additional structural properties of the Keccak permutation (namely, the limited diffusion of its internal mappings) in order to optimize cube testers. Finally, we devise a new key recovery method which also exploits the limited diffusion properties. Yet, the key recovery technique is based on a divide-and-conquer strategy, exploiting subtle interactions between the bits of the Keccak internal state, and is of independent interest.

The low algebraic degree of a Keccak round has been exploited in many previous attacks, and in particular, in key recovery attacks [21], preimage attacks [6] and zero-sum distinguishers on the permutation [2, 11]. However, most of those attacks (apart from [21]) were only applied to the non-keyed Keccak variants, whereas we focus on its keyed modes. Furthermore, several of these attacks seem to have limited applicability, as they either assume a very powerful attack model (which does not correspond to a realistic attack scenario), or give a marginal improvement over generic attacks. Compared to these related attacks, our attacks seem to have broader applicability (as they focus on concrete schemes that use the Keccak permutation), and are significantly more efficient than generic attacks.

From a methodological point of view, most related attacks [2, 6, 11, 21] were based on algebraic analysis of the Keccak non-linear component. Although such analysis can be highly non-trivial (e.g., see [11]), it mostly takes into account the low algebraic degree of the Keccak permutation, but ignores several other (potentially useful) properties of Keccak internal mappings. The main difference between our approach and the previous ones, is that we show how to combine structural properties of Keccak (such as the limited diffusion of its linear layer) in order to gain advantage over standard algebraic analysis. In fact, it is likely that some of our techniques may be combined with the previous attacks to improve their results.

The paper is organized as follows. In Section 2, we briefly describe the cube attack, and in Section 3, we describe the Keccak sponge function and its keyed variants we analyse. In Section 4 we present our cube attacks on Keccak. In Section 5 we describe attacks which are based on output prediction (exploiting cube testers). Finally, we present our divide-and-concur attack in Section 6 and conclude the paper in Section 7.

2 Cube Attacks

The cube attack is a chosen plaintext key-recovery attack, which was formally introduced in [14] as an extension of higher order differential cryptanalysis [20] and AIDA [23]. Since its introduction, the cube attack was applied to many different cryptographic primitives such as [3, 4, 21]. Below we give a brief description of the cube attack, and refer the reader to [14] 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 outputs 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*.*

A simple combinatorial proof of this theorem is given in [14]. 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 $f(v, x)$. We use the following notation

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

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, we can write

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

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 $a_i = \sum_{v\in C_t} f(v, 0, \dots, \underbrace{1}_{x_i}, 0, \dots, 0)) = a_i$

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, we exploit the ability of the attacker to choose 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 or in other words

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

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

$$a_1x_1 + \ldots + a_nx_n + c = b_t.$$

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 the Gaussian elimination.[1]

2.3 Cube Testers

The notion of cube testers was introduced in [3], as an extension of the cube attack. Unlike standard cube attacks, cube testers aim at detecting a 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.

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].

The sponge function works on a b-bit 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.

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.

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

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 the 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) {

$\theta$ 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)

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

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

$\chi$ 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]),

$\iota$ 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. Furthermore, χ only multiplies neighbouring bits (`A[x,y,z]` and `A[x+1,y,z]`). 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. In many cases it is useful to treat the state as the 5×5 array of 64-bit lanes. Each element of the array is specified by two coordinates, as shown in Figure 1.

[0,0]	[1,0]	[2,0]	[3,0]	[4,0]
[0,1]	[1,1]	[2,1]	[3,1]	[4,1]
[0,2]	[1,2]	[2,2]	[3,2]	[4,2]
[0,3]	[1,3]	[2,3]	[3,3]	[4,3]
[0,4]	[1,4]	[2,4]	[3,4]	[4,4]

Fig. 1: Lanes coordinates. Each square represents a lane in the state.

3.1 Keyed Modes of Keccak

The Keccak sponge function can be used in keyed mode, providing several different functionalities. Three of these functionalities which we analyse in this paper are a hash-based message authentication code (MAC), a stream cipher and an authenticated encryption (AE) scheme based on the design methods proposed in [10].

MAC based on Keccak A message authentication code (MAC) is used for verifying data integrity and authentication of a message. A secure MAC is expected to satisfy two main security properties. Assuming that an adversary has access to many valid message-tag pairs, (1) it should be infeasible to recover the secret key used and (2) it should be infeasible for the adversary to forge a MAC, namely, provide a valid message-tag pair (M, T) for a message M that has not been previously encrypted.

A hash-based algorithm for calculating a MAC involves a cryptographic hash function in combination with a secret key. A typical construction of such a MAC is HMAC, proposed by Bellare et al. [5]. However, for the Keccak hash function, the complex nested approach of HMAC is not needed and in order to provide a MAC functionality, we simply prepend the secret key to the message in order to calculate a tag. In this paper, we only use short messages such that the internal permutation is applied only once (see Figure 2).

Stream Cipher Based on Keccak In the stream cipher mode, the state is initialized with the secret key, concatenated with a public initialization vector (IV). After each Keccak permutation call, an r-bit keystream (where r is the bitrate of the Keccak instance) is extracted and used to encrypt a plaintext via bitwise XOR. In this paper, we only exploit the first r bits of keystream, such that the internal permutation is applied only once (as shown in Figure 3).

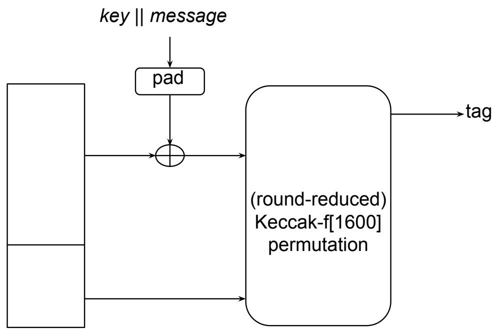

Fig. 2: MAC based on Keccak.

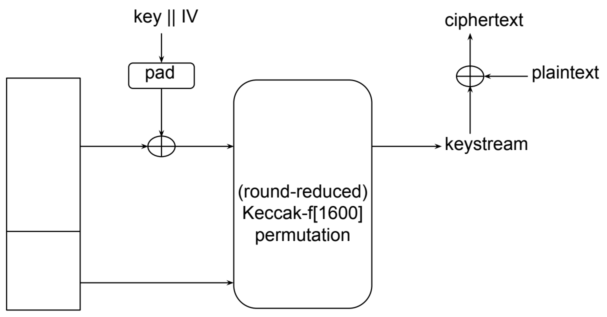

Fig. 3: Stream Cipher Based on Keccak

Authenticated Encryption Scheme Based on Keccak The Keccak sponge function can also be used as a building block for authenticated encryption (AE) schemes, simultaneously providing confidentiality, integrity, and authenticity of data. In this paper, we concentrate on the concrete design of Keyak [9] — a recently proposed authenticated encryption scheme, submitted to the CAESAR competition [1]. The scheme is based on the Keccak permutation with a nominal number of rounds set to 12.

Figure 4 shows the scheme of Lake Keyak, which is the primary recommendation of the Keyak family algorithms. In this scheme, the key and tag sizes are 128 bits long, and the capacity is set to $c = 252$ (i.e., $r = 1600 - 252 = 1348$). For a more formal description, we refer the reader to [9].

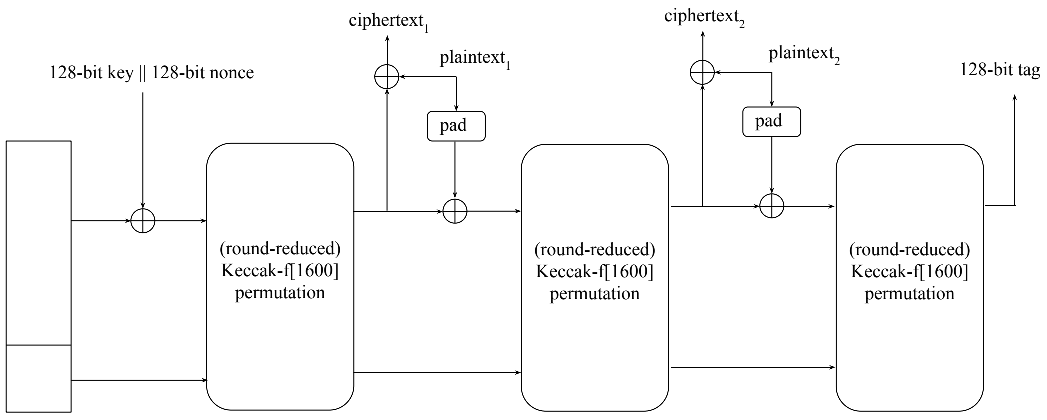
ciphertext$_1$
plaintext$_1$
ciphertext$_2$
plaintext$_2$
128-bit key || 128-bit nonce
128-bit tag
pad
pad
(round-reduced) Keccak-f[1600] permutation
(round-reduced) Keccak-f[1600] permutation
(round-reduced) Keccak-f[1600] permutation

Fig. 4: Lake Keyak processing two plaintext blocks.

Figure 4 shows how the Keyak scheme processes two plaintext blocks. The first permutation call of Keyak takes as an input a key and a nonce. We note that some of our attacks use up to 2 plaintext blocks, and further note that the scheme has an optional input of associated data, which we do not use.

According to the specification of Keyak, in order to assure confidentiality of data, a user must respect the nonce requirement. Namely, a nonce cannot be reused, otherwise, confidentiality is not guaranteed. However, for authenticity and integrity of data, a variable nonce is not required, and indeed some of our attacks in sections 6 and 5 reuse the nonce in order to break these properties.

3.2 Attack Models and Parameters

Most of our attacks focus on default variants of Keccak, suggested by the Keccak designers, namely 1600-bit state and 1024-bit bitrate, or a 1344-bit bitrate (used for example in Keyak and SHAKE-128). Furthermore, we concentrate on typical, real-life key, tag and IV lengths, avoiding artificial scenarios although they could potentially help the attacker (e.g., tags larger than 256 bits or very long IVs).

The Keccak sponge function can also work on smaller states, which may be useful for lightweight cryptography. In such Keccak variants, the size of the internal lanes is reduced and this has an effect on our attacks, as we highlight in several places.

All the attacks follow the chosen plaintext model, assuming that the attacker is able to choose various values of message/nonce/IV and obtain the corresponding ciphertext/tag/keystream outputs. For Keccak working as a MAC, we can control many input message bits, but only a short tag (128 or 256 bits) is available as an output. In the stream cipher mode the situation is reversed, as the attacker can only control IV bits (typically up to 256 bits), however, the output (keystream bits) can be as big as the bitrate (1024 bits for the default variant). Interestingly, for authenticated encryption mode (such as Keyak), we can take advantage of both long input and available output, as shown in Section 6.

4 Cube Attack on Keccak Sponge Function

In this section we focus on practical key recovery attacks which can be implemented and verified on a desktop PC. We analyse the round-reduced Keccak used as a MAC and a stream cipher. We also show how to recover the key for the 6-round Keyak.

4.1 Key Recovery 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 of the internal permutation is reduced to 5. The key and tag sizes are both 128 bits.

Preprocessing Phase 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 $2^5 = 32$ and for any cube with 31 variables, the superpoly consists of linear terms only. Thus, we can avoid the step of testing the superpolys for linearity in cube attacks. 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 pick 31 out of the 128 public variables and check whether the superpoly consists of any secret variables or it is a constant. With this simple strategy, we have been able to find 117 linearly independent expressions (superpolys) in a few days on a desktop PC (example is given in Appendix A). The search was somewhat more complex than expected, as only $20 - 25\%$ of the superpolys are useful (i.e., non-constant). On the other hand, we found more superpolys and shortened the search time by examining different output bits (with their corresponding superpolys).

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 the 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 cost of linear algebra can be neglected).

4.2 Key Recovery 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 a specific property the Keccak χ step. 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 given output bits. Consequently, the final nonlinear step χ can be inverted and the cube attack is reduced to 5.5 rounds. As the first half of a round is linear and does not increase the degree, the output bits have a manageable polynomial degree of at most 32 and the scenario is very similar to the one considered in the previous attack.

Standard MACs are of size 128 or 256 output bits, which are insufficient for inversion — these output bits do not allow us to uniquely calculate any bit (or a linear combination of bits) after 5.5 rounds. If we consider longer MACs (for instance with 320 bits), then the attack setting becomes 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. A good example is Keccak used as a stream cipher, and here, we attack the default variant of Keccak with 1600-bit state, $r = 1024$, $c = 576$ with key and IV sizes of 128 bits. The first 960 of the 1024 available output bits contain $960/5 = 192$ full rows (each sequence of 320 bits contains $320/5 = 64$ full rows), which can be inverted and exploited in the attack.

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 (example is given in Appendix B). This gives an online attack complexity of $2^{31} \cdot 25 \cong 2^{36}$.

4.3 Key Recovery Attack on MAC-based State-reduced 6-round Keccak

We attack the Keccak MAC that operates on a 400-bit state with an 80-bit key. As the state is smaller, 128 bits of MAC (output bits) cover all the rows in the state and we are able to invert these rows through the ι and χ steps. Therefore, our attack on the 6-round Keccak MAC becomes practical.

During the preprocessing phase, we have found 80 linearly independent superpolys using 18 cubes. This allows us 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 when the state is smaller. An example of a cube chosen for the attack is given in Appendix C.

4.4 Key Recovery Attack on 6-round Keyak

The key recovery attack is essentially the same as the one described for the stream cipher mode. Instead of IV variables, we use the nonce as cube variables. After a single call to the Keccak permutation, the bitrate part of state is available (by XORing known plaintext with the corresponding ciphertext – see Figure 4). As in the stream cipher mode, we have many output bits available ($r = 1348$), allowing to easily invert ι and χ and break 6 rounds.

5 Output Prediction for Keyed Variants of Keccak

In this section, we first present a practical cube tester for 6.5-round Keccak, and then show how to exploit similar distinguishers in order to predict the output of Keccak when used as a MAC (i.e., mount a forgery attack) or in stream cipher mode.

5.1 Practical Cube Tester for 6.5-round Keccak Permutation

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 (without exploiting some internal invertibility properties, as in the previous section). However, if we carefully choose the cube variables, we can exploit a special property of θ in order to considerably reduce the output degree after 6 rounds and keep the complexity low.

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

We choose the 33-dimensional cube $\{v_0, v_1, \ldots, v_{32}\}$ such that $v_i = A[0, 2, i]$, while ensuring that the column parities remain constant by setting the additional linear constraints $A[0, 3, i] = v_i \oplus c_i$, for arbitrary binary constants c_i (see Figure 5). In order words, we sum over the outputs of the 33-dimensional linear subspace defined on the 66 state bits $A[0, 2, i], A[0, 3, i]$ by 33 equations $A[0, 2, i] = A[0, 3, i] \oplus c_i$ for $i \in \{0, 1, \ldots, 32\}$. The remaining bits of the input state (some of which are potentially unknown secret variables) are set to arbitrary constants. As can be seen from Figure 5, at the input to χ, each 5-bit row of the state contains at most one variable, and therefore, the variables are not multiplied together in the first round as required.

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-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.

Assume that the 33-dimensional cube is missing an output for one value of the public variables. Then, as the sums of all 2^{33} outputs is zero, the missing output is equal to the sum of the remaining $2^{33}-1$ outputs. Thus, the distinguishing attack can be used to predict the output of the cipher for a previously unseen input value. In the rest of this section, we exploit this property in more specific attacks on keyed modes of Keccak.

We note that it possible to predict the values of several outputs at a smaller amortized cost than summing over a 33-variable cube for each missing value. This can be accomplished by using larger cubes which contain several missing outputs, and using more complex algebraic algorithms (such as the ones used in [6]) in order to calculate them. However, in this paper we focus on basic attacks that predict only one output.

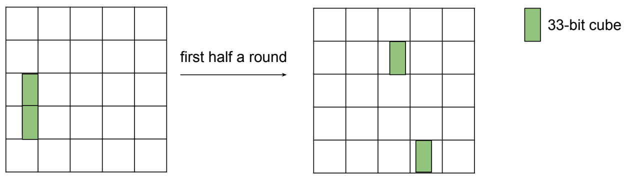

Fig. 5: The initial state of a cube tester and the transition through the first linear part of the round (θ, ρ, π steps)

5.2 Extending the Cube Tester to Smaller States and More Rounds

When considering Keccak variants with states of $b = 400$ bits, then each lane contains (at most) 16 bits, and it is not clear how to select the 33-variable cube as in the previous case. However, we can generalize the above idea by noticing that it is possible to carefully select a cube of dimension of (up to) $4 \cdot 16 = 64$ such that its variables are not multiplied together in the first round. Such a cube contains (up to) 64 independent variables in 4 lanes of 16 columns, where the 5th lane keeps the columns parities constant (e.g., to keep the column parities to zero, its value has to be equal to the XOR of the 4 independent lanes). One can observe that after the application of ρ and π, these variable are not multiplied together by χ in the first round.

A careful selection of cube variables allows to select large cubes for which the complexity of the distinguishing attack is significantly reduced compared to the complexity with arbitrary cubes. We note that there exist other methods to carefully select a large set of variables that are not multiplied together in the first round, and we only give one of these methods in this section.

Clearly, the idea can also be exploited for Keccak variants with larger states, for which we can select larger cubes and mount distinguishing attacks on more than 6.5 rounds. In such cases, the attack becomes impractical, but it may still be more efficient than generic attacks (depending on the attack setting).

5.3 MAC Forgery for 7-round Keccak MAC and Keyak

We attack 7-round Keccak MAC with the default bitrate $r = 1024$ and 128-bit key and tag length, ideally providing 128-bit security. In order to forge a tag T for an arbitrary message M, we use a cube tester similar to the one from Section 5.1. We choose a 65-variable cube, as shown in Figure 6. The remaining bits of the input state are set according to the message bits of M which are not part of the cube and additional constants (some of which are potentially unknown secret variables). Due to the placement of the cube variables, we can go through the first round without increasing an algebraic degree. Consequently, after 7 rounds, the degree is at most 64 (rather than 128). Therefore, the cube sum of any output bit after 7 rounds is equal to zero.

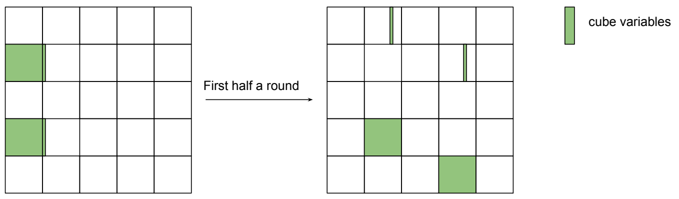

Fig. 6: The transition through the first half a round for the cube tester exploited in the MAC forgery attack. In the initial state cube variables are assumed to be equal column-wise.

The forgery attack works by collecting $2^{65} - 1$ tags for chosen messages which consist of all 2^{65} messages defined by the cube (shown in Figure 6), with the exception of M. Since the cube sums of all 128 output bits of the MAC are zero, we can calculate the tag of M by XORing the $2^{65} - 1$ known tags. Therefore, we forge a valid message-tag pair (M, T), which has not been previously seen.

MAC Forgery for 7-round Keyak For Keyak, the selection of cube variables is limited to the 128 nonce bits, and it is not clear how to exploit the method above to gain an extra round. However, according to the Keyak specification, a variable nonce is not required for authenticity and integrity of data. Therefore, we fix the nonce, and hence also fix the state after the first permutation call. Then, before the second permutation call, the state absorbs plaintext variables, which we select as cube variables. In this setting, we can control as many as $r = 1348$ state bits, and forge the 128-bit tag with complexity 2^{65}, similarly to the 7-round Keccak MAC forgery above.

5.4 MAC Forgery for 8 rounds

We attack 8-round variants with a longer 256-bit key and tag, increasing the security level to 256 bits. The other parameters remain the same. In this case, we select a 129-variable cube as follows: 128 variables among lanes $A[4,0], A[4,1], A[4,2]$ (as $c = 576$, these lanes contain public message bits), using the generalized idea of Section 5.2, and 1 additional variable in lanes $A[2,1], A[2,2]$. After the linear layer, these variables diffuse to lanes $A[2,0], A[1,1], A[1,2], A[0,3], A[2,4]$ which have different y indices and are not multiplied together in the first round. Therefore, for such a selection, the output degree in the variables after 8 rounds is at most $2^7 = 128$, implying that the cube sums are zero, and we can forge a message with complexity 2^{129}.

5.5 Keystream Prediction for 8- and 9-round Keccak-based Stream Cipher

The output prediction strategy used to forge a MAC can be used to predict the keystream of a previously unseen IV in the stream cipher mode. The difference is that in this mode we generally have less control over the public variables (IV), and we cannot select the cube variable as in the previous attacks to gain a round at the beginning. On the other hand, we exploit larger cubes than in MACs (as some stream ciphers aim for higher security level compared to MACs). Furthermore, as more output bits are generally available, we can invert the last non-linear χ on sequences of 320 output bits and reduce their algebraic degree as in Section 4.

First, let us describe the attack on the 8-round variant with the default parameters $r = 1024$ and $c = 576$. We set the key length to 256 bits and IV length to 128 bits. Having 1024 bits of keystream, we can invert as many as 960 bits through ι and χ. Therefore, we reduce our attack to 7.5 rounds for which the algebraic degree is at most $2^7 = 128$. The attack has two phases.

Preprocessing Phase

Since for 7.5 rounds we deal with the algebraic degree 128, summing over any 128-bit cube gives a constant regardless of the secret key values. Thus, we determine a cube sum (either 1 or 0) for each output bit of the first $3 \cdot 320 = 960$ bits of state (which can be fully inverted online, given the 1024-bit keystream after 8 rounds). The cost of the preprocessing is 2^{128} Keccak calls.

Online Phase

Keystream prediction for an unused IV follows the same pattern as for a MAC forgery. First, we collect $2^{128} - 1$ keystream sets for IVs which reside in a 128-dimensional cube (not including the IV whose keystream we predict), and invert ι and χ on the first available bits $3 \cdot 320 = 960$ bits of state. Then, the first 960 bits of keystream of the remaining IV can be predicted. This is done by XORing the 960 bits of all the $2^{128} - 1$ inverted keystreams to the cube sums calculated during preprocessing, and then reapplying χ and ι to the outcome. The complexity of the online phase is 2^{128} Keccak calls, whereas the generic attack complexity is 2^{256}.

Keystream Prediction for 9-round Keccak

Following the procedure from 8-round variant, we can easily extend the attack to 9-round Keccak with $r = 1024, c = 576$ using larger keys of 512 bits, and an IV of 256 bits. The only difference is that we would sum over larger 256-bit cubes. According to the recent analysis [18], the ideal security level of this variant should be 512 bits[2], whereas our attack has complexity of 2^{256}.

6 Divide-and-Conquer Key Recovery Attack on Keccak-based MAC and Keyak

In the previous section, we showed how to predict output for several keyed variants of Keccak. In this section, we return to the most powerful type of attacks, and describe key recovery attacks on the 6- and 7-round Keccak with a 1600-bit state. We attack a variant with the capacity parameter $c = 256$ and a 128-bit key. Therefore, the security level of this variant is 128 bits.

First, we note that for 6 rounds the degree of the output bits is generally $2^6 = 64$, and thus the preprocessing phase of the standard cube attack is too expensive to perform (without exploiting some internal invertibility properties, as in Section 4). Another possible approach it to carefully select the cube such that we obtain a practical distinguisher (as in Section 5.1). Then, we can try to apply several techniques that were developed to exploit similar distinguishers for key recovery (such as conditional differential cryptanalysis [19] and dynamic cube attacks [15]). However, these techniques seem to be better suited for stream ciphers built using feedback shift registers, rather than the SP-network design of Keccak.

[2] According to [18], the security of the scheme is $min(n, c, b/2) = min(512, 576, 800) = 512$ bits.

As it is not clear how to use the standard key recovery techniques in our case, we use a different approach. The main idea in our attack is to select the public variables of the cube in such a way that the superpolys depend only on a (relatively) small number of key bits, whose value can be recovered independently of the rest of the key. Thus, the full key can be recovered in several phases in a divide-and-conquer manner.

The approach we use is similar to the one used in the attacks on the stream ciphers Trivium and Grain in [16]. However, while the results on Trivium and Grain were mostly obtained using simulations, our attack is based on theoretical analysis that combines algebraic and structural properties of Keccak in a novel way. This analysis enables us to estimate the complexity of the attack beyond the feasible region (in contrast to the simulation-based attack of [16]).

Borderline Cubes The starting point of the attack is the cube tester of Section 5.1, which is based on a 33-variable cube, whose column parities remain constant for all of their 2^{33} possible values. As the cube variables are not multiplied together in the first round and the degree of 6-round Keccak in the state variables after one round is $2^5 = 32$, then the cube sums for the 33-variables are zero for all output bits. When we remove one variable from this cube, the sums are no longer guaranteed to be zero and they depend on the values of some of the constant bits of the state. This leaves us in a borderline situation. If a state bit is not multiplied with the cube variables in the first round, then the cube sums do not depend on the value of this bit. On the other hand, if a state bit is multiplied with the cube variables in the first round, then the cube sums generally depend on the value of the bit (assuming sufficient mixing of the state by the Keccak mapping). Thus, by a careful selection of a “borderline” cube of dimension 32, we can assure that the cube sums depend only on a (relatively) small number of key bits. This gives rise to divide-and-conquer attacks, which we describe in detail in the rest of this section.

6.1 Basic 6-Round Attack

According to the Keccak MAC specification, the 128-bit key is placed in $A[0,0]$ and $A[1,0]$. However, it is worth noting that our attack could be easily adapted to any other placements of the secret key. We select 32 cube variables $v_1, v_2, \ldots, v_{32}$ in $A[2,2]$ and $A[2,3]$, such that the column parities of $A[2,*]$ remain constant for the 2^{32} possible values of the variables (similarly to Section 5.1). This careful selection of the cube variables leads to two properties on which our attack is based:

Property 1. The cube sum of each output bit after 6 rounds does not depend on the value of $A[1,0]$.

Property 2. The cube sums of the output bits after 6 rounds depend on the value of $A[0,0]$.

The detailed proof of these properties is given in Appendix D, but note that as we selected a “borderline” cube of 32 variables, we can prove Property 1 by showing that the cube variables are not multiplied with the secret variables of $A[1,0]$ in the first round. Similarly, we can prove Property 2 by showing that the cube variables are multiplied with the secret variables of $A[0,0]$ in the first round.

We now describe the attack which exploits the two properties to retrieve the value of $A[0,0]$. For the sake of convenience, we separate the attack to preprocessing and online phases, where the preprocessing phase does not depend on the online values of the secret key. However, we take into account both of the phases when calculating the complexity of the full attack. The preprocessing phase is described below.

1. Set the capacity lanes ($A[1,4]$, $A[2,4]$, $A[3,4]$, $A[4,4]$) to zero. Set all other state bits (beside $A[0,0]$ and the cube variables) to an arbitrary constant.[3]

[3] The chosen constant has to include padding bits.

2. For each of the 2^{64} possible values of $A[0,0]$:
   (a) Calculate the cube sums after 6 rounds for all the output bits. Store the cube sums in a sorted list L, next to the value of the corresponding $A[0,0]$.

As the cube contains 32 variables, the time complexity of Step 2.(a) is 2^{32}. The cube sums are calculated and stored for each of the 2^{64} values of $A[0,0]$, and thus the total time complexity of the preprocessing phase is $2^{64} \cdot 2^{32} = 2^{96}$, while its memory complexity is 2^{64}.

The online phase, which retrieves $A[0,0]$, is described below.

1. Request the outputs for the 2^{32} messages that make up the chosen cube (using the same constant as in the preprocessing phase).
2. Calculate the cube sums for the output bits and search them in L.
3. For each match in L, retrieve $A[0,0]$ and store all of its possible values.

Although the actual online value of $A[1,0]$ does not necessarily match its value used during preprocessing, according to Property 1, it does not affect the cube sums. Thus, we will obtain a match in Step 3 with the correct value of $A[0,0]$. In order to recover $A[1,0]$, we independently apply a similar attack using 32 public variables in $A[4,2]$ and $A[4,3]$ (for which properties corresponding to Property 1 and Property 2 would apply). Finally, in order to recover the full key, we enumerate and test all combinations of the suggestions independently obtained for $A[0,0]$ and $A[1,0]$.

The time complexity of the attack depends on the number of matches we obtain in Step 3. The expected number of matches is determined by several factors, and in particular, it depends on a stronger version of Property 2, namely on the actual distribution of the cube sums after 6 rounds in $A[0,0]$ (Property 2 simply tells us that the distribution is not concentrated in one value of $A[0,0]$). Furthermore, the number of matches varies according to the number of available output bits, and the actual cube and constants chosen during preprocessing Step 1 (and reused online). In general, assuming that the cube sums are uniformly distributed in $A[0,0]$, and we have at least 64 available output bits (which is the typical case for a MAC), we do not expect more than a few suggestions for the 64-bit $A[0,0]$ in online Step 3. Although we cannot make the very strong assumption that the cube sums are uniformly distributed in $A[0,0]$, our experiments (described in Appendix D) indeed reveal that we are likely to remain with very few suggestions for $A[0,0]$ in online Step 3. Furthermore, even if we remain with more suggestions than expected, we can collect sums from several cubes, obtained by choosing different cube variables or changing the value of the message bits, which do not depend on the cube variables. This reduces the number of matches in Step 3 at the expense of slightly increasing the complexity of the attack. We thus assume that the number of matches we obtain in Step 3 is very small.

The online phase requires 2^{32} data to retrieve the 64-bit $A[0,0]$ and requires 2^{32} time in order to calculate the cube sums. As previously mentioned, in order to recover $A[1,0]$, we independently apply the same attack but this time using 32 public variables in $A[4,2]$ and $A[4,3]$. Thus, for the full key recovery, the total data complexity is 2^{33} and the online time complexity is 2^{33} (assuming that we do not have too many suggestions in Step 3). Taking preprocessing into account, the total time complexity is 2^{96}, and the memory complexity is 2^{64}.

6.2 Balanced 6-Round Attack

The basic attack above employs an expensive preprocessing phase which dominates its time complexity. In this section, we describe how to tradeoff the complexity of the preprocessing and online phases, allowing us to devise a more efficient attack.

The imbalance of the basic attack comes from the fact that the cube sums after 6 rounds depend on all the variables of $A[0,0]$. Thus, we need to iterate over all of their 2^{64} values during preprocessing to cover all the possible values of the cube sums in the online phase.

In order to reduce the preprocessing complexity, we aim to eliminate the dependency of the cube sums on some of the variables of $A[0,0]$. However, it is not clear how to achieve this, as we cannot control the values of the secret variables and thus we cannot directly control their diffusion. On the other hand, as the action of θ is only determined by the column parities, we can indirectly control the diffusion of the secret variables by using additional *auxiliary variables* in the lanes with $x = 0$, and specifically in $A[0,1]$. If we set the column parities for $x = 0$ to zero (or any other pre-defined constant), then the diffusion of the secret key is substantially reduced. Figure 7 shows an impact of the auxiliary variables on diffusion of the secret key variables.

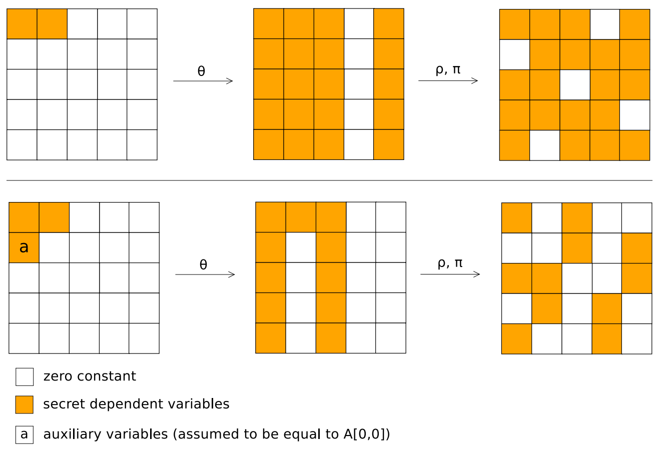
θ
ρ, π
a
θ
ρ, π
zero constant
secret dependent variables
a
auxiliary variables (assumed to be equal to A[0,0])

Fig. 7: Impact of auxiliary variables on diffusion of the secret key variables. When using auxiliary variables, $A[0,0]$ (secret variables) and $A[0,1]$ (auxiliary variables) are diffused to $A[0,0]$ and $A[1,3]$, without affecting many lanes of the state.

Similarly to the basic attack, we select a borderline cube with 32 variables in $A[2,2]$ and $A[2,3]$, such that the column parities of $A[2,*]$ remain constant for the 2^{32} possible values of the variables. As explicitly shown in the proof of Property 1 in Appendix D, the cube variables are not multiplied with the auxiliary variables or secret variables in the first round (assuming that the column parities of $x = 0$ are fixed). Therefore, the cube sums after 6 rounds depend neither on the value of $A[0,0]$, nor on the auxiliary variables of $A[0,1]$ (but only on the column parities of $x = 0$). This observation gives rise to our balanced attack. Similarly to the basic attack, we divide the attack into preprocessing and online phases, where the preprocessing phase is described below.

1. Set the state bits (which are not cube variables) to zero (or an arbitrary constant). Furthermore, set $A[1,0]$ and the 32 LSBs of $A[0,0]$ to zero (or an arbitrary constant).
2. For each possible value of the 32 MSBs of $A[0,0]$:
   (a) Calculate the cube sums after 6 rounds for all the output bits. Store the cube sums in a sorted list L, next to the value of the 32 MSBs of $A[0,0]$.

The cube sums are calculated and stored for each of the 2^{32} values of the 32 MSBs of $A[0,0]$, and thus the total time complexity of the preprocessing phase is $2^{32} \cdot 2^{32} = 2^{64}$, while its memory complexity is 2^{32}.

The online phase is described below.

1. For each possible value of the 32 LSBs of $A[0,1]$:
   (a) Request the outputs for the 2^{32} messages that make up the chosen cube with the 32 LSBs of $A[0,1]$ set according to Step 1 (setting the same constant values in the state as in the preprocessing).

(b) Calculate the cube sums for the output bits and search them in L.
(c) For each match in L, retrieve the 32 MSBs of $A[0,0]$. Assume that the 32 LSBs of $A[0,0]$ are equal to the 32 LSBs of $A[0,1]$ (the 32 column parities should be zero, as in the preprocessing phase). Then, given the full 64-bit $A[0,0]$, exhaustively search $A[1,0]$ using trial encryptions, and if a trial encryption succeeds, return the full key $A[0,0], A[1,0]$.

Once the value of the 32 LSBs of $A[0,1]$ in Step 1 is equal to the 32 LSBs of the (unknown) $A[0,0]$, the corresponding column parities are zero, and thus they match the column parities assumed during preprocessing. The actual values of the 32 LSBs of $A[0,1]$ and $A[0,0]$ (and the actual value of $A[1,0]$) do not necessarily match their values during preprocessing. However, they do not influence the cube sums, and thus the attack recovers the correct key once the value of the 32 LSBs of $A[1,0]$ is equal to the 32 LSBs of $A[0,0]$.

The online phase requires $2^{32+32} = 2^{64}$ chosen messages to retrieve the 64-bit $A[0,0]$. Assuming that we do not have too many suggestions in Step 3 (as assumed in the basic attack), it requires 2^{64} time in order to obtain the data and calculate the cube sums, and additional 2^{64} time to exhaustively search $A[1,0]$ in Step 1(c). Taking preprocessing into account, the total time complexity of the attack is about 2^{66}, and its memory complexity is 2^{32}.

We note that it is possible to obtain additional tradeoffs between the preprocessing and online complexities by adjusting the number of auxiliary variables. However, in this paper we describe the attack with the best total time complexity only.

6.3 7-Round Attack

For a MAC based on the 7-round Keccak, the algebraic degree of the output in the state variables of round 1 is $2^6 = 64$. We can extend our 6-round attack to 7 rounds by selecting a borderline cube of 64-variables (i.e., a full lane) in $A[2,2]$ and $A[2,3]$. As the cube consists of 32 more variables than the cube of the 6-round attack, the data complexity increases by a factor of 2^{32}, and the time complexity of both the preprocessing and online phases increases by the same factor (except for the exhaustive search for $A[1,0]$ in online Step 1(c), which still requires 2^{64} time). Thus, the data complexity of the full 7-round attack is 2^{64}, its time complexity is 2^{97}, and its memory complexity remains 2^{32}.

Comparison with Standard Cube Attacks One may claim that the 6-round attacks presented in this section are somewhat less interesting, as it seems reasonable that the standard cube attack (such as the ones presented in Section 4) would break the scheme in a similar time complexity of (a bit more than) $2^{2^6} = 2^{64}$. However, in order to mount the standard cube attack, we need to run a lengthy preprocessing phase whose outcome is undetermined, whereas our divide-and-conquer algorithm is much better defined. Furthermore, the divide-and-conquer attacks allow a wider range of parameters and can work with much less than 2^{64} data.

Despite its advantage in attacking 6 rounds, the real power of the divide-and-conquer attack introduced in this paper is demonstrated by the 7-round attack. Indeed, the standard cube attack on the 7-round scheme is expected to require more than $2^{2^7} = 2^{128}$ time, and is therefore slower than exhaustive search, whereas our divide-and-conquer attack breaks the scheme with complexity 2^{97}.

6.4 Application to 7-Round Keyak

We now apply the divide-and-concur attack to 7-round Keyak. As in the forgery attack on 7-round Keyak of Section 5.3, we reuse the nonce and consider the message bits as public variables in order to have more freedom and gain an additional round at the beginning. Therefore, we only aim to break the authenticity and integrity of Keyak. Compared to the attack of Section 5.3 which allows to forge a single tag, the 7-round attack described here is significantly stronger. This attack recovers the secret key, after which the security of the system is completely compromised (e.g. one can immediately forge the tag of any message).

In the initial setting, all the 1600 state bits obtained after the first permutation are unknown, and we aim to recover them. Once this state is recovered, we can run the permutation backwards and recover the secret key. In order to recover the secret 1600-bit state, we first obtain the encryption of an arbitrary 2-block message whose first-block ciphertext reveals the value of $r = 1348$ bits of secret state. Then, during the actual attack, we choose messages that set these $r = 1348$ known bits to an arbitrary pre-fixed constant (e.g., zero), whose value is defined and used during the preprocessing phase of the attack. Using this simple idea, the number of secret state variables for Keyak is reduced from 1600 to $c = 252$ variables in $A[1,4], A[2,4], A[3,4], A[4,4]$. Note that although the key size is only 128 bits, we have a larger number of 252 secret variables.

In this attack, we use a borderline cube containing $d = 32$ variables. Recall that in the case of MAC-based Keccak, a 32-variable cube was used to attack 6 rounds, but here, we have a larger output of 1348 bits which allows to exploit the inversion property from Section 4. Therefore, we can attack 7 rounds using a 32-bit borderline cube, exploiting the inversion property on $320 \cdot 4 = 1280$ output bits.

In the attack on the 6-round Keccak MAC, we selected the 32 cube variables by varying 64 bits in $A[2,2]$ and $A[2,3]$, which diffuse to different 64 bits after the linear layer. As χ only multiplies consecutive bits in a row, then each such bit is multiplied with 2 neighbouring bits in its row, and therefore the 64 bits are multiplied with 128 bits that remain constant during the cube summation. As we selected a borderline cube, these 128 constant bits are the only ones that effect the value of the cube summations (which is the crucial property on which the divide-and-conquer attack is based), and we refer to these bits here as *effective bits.* Some of the values of the 128 effective bits are unknown as they depend on linear combinations of secret variables (such a combination can either be a singleton bit, or a linear combination of several secret bits), which we refer to here as *effective secret expressions.* Note that since each effective bit contains at most one effective secret expression, then the number of effective secret expressions is upper bounded by the number of effective bits.

In the case of the 6-round attack on the Keccak MAC, only 64 of the 128 effective bits actually depend on secret material (i.e., the number of effective secret expressions is 64). In order to recover the 64 bits of effective secret expressions, the idea was to enumerate their values during preprocessing, store their cube sums, and compare these sums to the ones obtained online. Therefore, the complexity of the basic (non-balanced) attack was about $2^{64+32} = 2^{96}$.

In the case of 7-round Keyak, we have as many as 252 secret variables, which extensively diffuse over the state. Therefore, a selection of a cube similar to the 6-round attack on MAC will cause the 64 cube variables to be multiplied with (the maximal number of) 128 effective secret expressions (instead of 64) in the first round, increasing the complexity of the basic attack to about $2^{128+32} = 2^{160}$, much above the exhaustive search of 2^{128}. In order to reduce the number of effective secret expressions, we use the idea from Section 5.2, and choose the 32 cube variables among the 5 lanes with $x = 0$. More precisely, we set the 8 LSBs of the first 4 lanes $A[0,0], A[0,1], A[0,2], A[0,3]$ as independent cube variables (i.e., we have a total of $4 \cdot 8 = 32$ independent variables), while the 8 LSBs of $A[0,4]$ act as "parity checks". Using that selection of cube variables, we have only 40 bits (instead of 64) that depend on the cube variables. The first linear layer diffuses these 40 bits to $A[0,0], A[2,1], A[4,2], A[1,3], A[3,4]$. These lanes have distinct y coordinates, and are therefore not multiplied together by χ — the condition to get the first round for 'free'.

Once again, as χ multiplies each bit with two neighbouring bits in its row, the 40 bits that depend on the cube variables are multiplied with $40 \cdot 2 = 80$ effective bits, which implies that the number of effective secret expressions is at most 80. Figure 8 shows an example diffusion of the 40-bit cube and the placement of the effective bits.

We now use the same procedure that we used for the basic attack on 6-round MAC to recover the values of the 80 effective secret expressions. Namely, during the preprocessing phase, we enumerate and

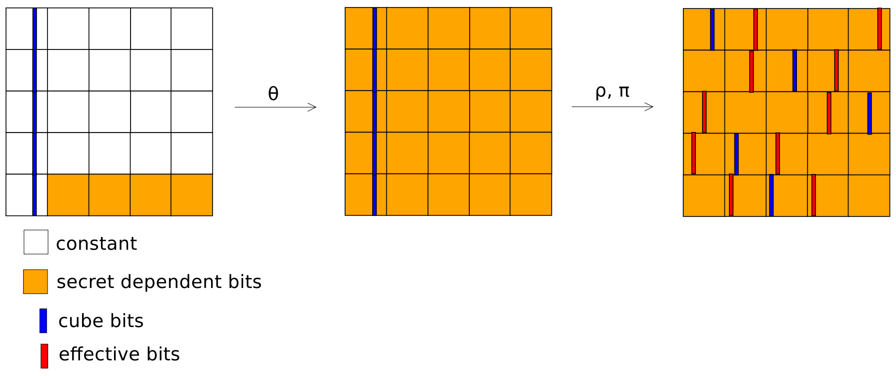

Fig. 8: Example placement of the cube and effective bits before the first χ is applied.

store the 2^{80} cube sums for all the possible values of the secret expressions in time $2^{80+32} = 2^{112}$, using 2^{80} memory. During the online phase, we simply request the outputs for the chosen cube, calculate the cube sums and compare with the values stored in memory. This online procedure recovers the secret expressions in 2^{32} data and time.

In order to recover all the 252 secret variables, we use a total of 8 cubes, obtained by rotating the variables of the initial cube inside the lanes by multiples of 8 towards the MSB (e.g., the second cube contains bits 8–16 of the lanes with $x = 0$). Each such cube changes the effective secret expressions that are multiplied with the cube variables (although their number remains 80). One can verify that the secret expressions multiplied with these 8 cubes contain sufficient information to recover all the 252 secret variables (by solving a system of linear equations in the 252 variables). Note that as we only have 252 secret variables, after exploiting the first few cubes, the values of some secret linear expressions is already determined, and this can be used to slightly optimize the attack.

Balanced Attack As in the case of the Keccak MAC, we can use auxiliary variables to balance the preprocessing and online complexities, reaching the lower total complexity. In the preprocessing, we calculate and store only 2^{40} cube sums — a substantially smaller subset of all possible 2^{80} cube sums for the 80 effective secret expressions. Thus, the preprocessing time complexity is reduced to $2^{40+32} = 2^{72}$ and the memory complexity is reduced to 2^{40}.

During the online phase, we exploit the large freedom in the message bits (we have $1348 - 40 - 2 = 1306$ free message bits that are not cube variables or padding bits) to set auxiliary variables that affect the values of the 80 effective bits which are multiplied with the cube variables. Then, we request the plaintexts and calculate online the cube sums for 2^{40} (unknown beforehand) different values of these 80 effective bits. According to the birthday paradox, with high probability, the (unknown beforehand) values of the 80 effective bits, in one of these online trials, will match one of their 2^{40} preprocessed values. This match will be detected by equating the cube sums, and allow us to recover the 80 effective secret expressions. Therefore, the data and time complexities of recovering 80 effective secret expressions are $2^{32+40} = 2^{72}$, and including preprocessing, the total time complexity is $2 \cdot 2^{72} = 2^{73}$.

In order to recover all the 252 secret variables, we use the 8 cubes defined in the basic (non-balanced) attack. Therefore, the total time complexity of the attack is $8 \cdot 2^{73} = 2^{76}$, the data complexity is $8 \cdot 2^{72} = 2^{75}$ and it requires $8 \cdot 2^{40} = 2^{43}$ words of memory (the memory can be reduced to 2^{40} if the cubes are analysed sequentially).

There are many possible tradeoffs between complexities of the preprocessing and the online phase. Interesting parameters are obtained by using only 24 auxiliary variables. In this case, according to the birthday paradox, we need to iterate over $2^{80-24} = 2^{56}$ values of the effective secret expressions for each cube during the preprocessing. Thus, the preprocessing complexity is $8 \cdot 2^{32+56} = 2^{91}$, the memory complexity is $8 \cdot 2^{56} = 2^{59}$, while the data and online time complexities are $8 \cdot 2^{32+24} = 2^{59}$ as well.

7 Conclusion

We mounted various types of algebraic attacks on keyed Keccak variants, breaking up to 6 rounds with practical complexity, and up to 9 rounds much faster than the exhaustive search. Our attacks incorporate in a novel way both algebraic and structural analysis of Keccak. We expect that the techniques developed in this paper will be further refined and used in future analysis of Keccak and related designs.

Considering attacks that break core security properties of the keyless, hashing mode, much faster than exhaustive search, the best result is 5 rounds [13]. As we can break up to 9 rounds of the keyed variants, the conclusion from our analysis is that the security margin of Keccak is somewhat reduced in the keyed modes. However, the full 24-round variants still have a big security margin. For Keyak – the authenticated encryption scheme based on Keccak – the nominal number of rounds is 12 and we showed that its security margin is smaller (but still sufficient).

References

1. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, `http://competitions.cr.yp.to/caesar.html`
2. 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)
3. 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)
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. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: Keyak, `http://keyak.noekeon.org`
10. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Duplexing the sponge: single-pass authenticated encryption and other applications. Cryptology ePrint Archive, Report 2011/499 (2011), `http://eprint.iacr.org/`
11. Boura, C., Canteaut, A., Cannière, C.D.: Higher-Order Differential Properties of Keccak and *Luffa*. In: Fast Software Encryption - 18th International Workshop, FSE 2011, Lyngby, Denmark, February 13-16, 2011, Revised Selected Papers. pp. 252–269 (2011)
12. 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)
13. Dinur, I., Dunkelman, O., Shamir, A.: Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials. In: Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers. pp. 219–240 (2013)
14. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. In: EUROCRYPT. pp. 278–299 (2009)
15. Dinur, I., Shamir, A.: Breaking Grain-128 with Dynamic Cube Attacks. In: FSE. pp. 167–187 (2011)
16. Fischer, S., Khazaei, S., Meier, W.: Chosen IV Statistical Analysis for Key Recovery Attacks on Stream Ciphers. In: Vaudenay, S. (ed.) AFRICACRYPT. Lecture Notes in Computer Science, vol. 5023, pp. 236–245. Springer (2008)
17. 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)
18. Jovanovic, P., Luykx, A., Mennink, B.: Beyond $2^{c/2}$ security in sponge-based authenticated encryption modes. Cryptology ePrint Archive, Report 2014/373 (2014)

19. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional Differential Cryptanalysis of NLFSR-Based Cryptosystems. In: ASIACRYPT. pp. 130–145 (2010)
20. 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)
21. Lathrop, J.: Cube Attacks on Cryptographic Hash Functions. Master's thesis, Rochester Institute of Technology (2009)
22. 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)
23. Vielhaber, M.: Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack. Cryptology ePrint Archive, Report 2007/413 (2007)

Appendix

A

Table 2: Example of cube 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		

B

Table 4: Example of cube 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		

C

Table 6: Example of cube 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		

D

In this section, we provide detailed analysis of some elements of the divide-and-conquer attack of Section 6.

Proofs of Properties 1 and 2

We prove the two properties on which the basic attack of Section 6 is based. Recall that we select the cube variables $v_1, v_2, \ldots, v_{32}$ in $A[2,2]$ and $A[2,3]$, such that the column parities remain constant for all the 2^{32} possible values of the variables.

Property 1 (restated). *The cube sum of each output bit after 6 rounds does not depend on the value of* $A[1,0]$.

Proof. We fix the value of $A[0,0]$ to an arbitrary constant, and symbolically represent the 64 bits of $A[1,0]$ as secret variables. We track the symbolic evolution of the 64 secret variables and 32 public variables throughout the first round: Due to θ, the secret variables of $A[1,0]$ linearly diffuse to $A[0,*]$, $A[2,*]$ and $A[1,0]$, while the cube variables of $A[2,2]$ and $A[2,3]$ do not diffuse (as the column parities of $A[2,*]$ remain constant). Then, ρ rotates the lanes, but does not effect the inter-lane diffusion. The mapping π reorders the lanes, and after its application, the bits of the 2 lanes $A[2,0], A[3,3]$ linearly depend on the public variables, while the bits of the following 11 lanes linearly depend on the secret variables. Figure 9 shows the details.

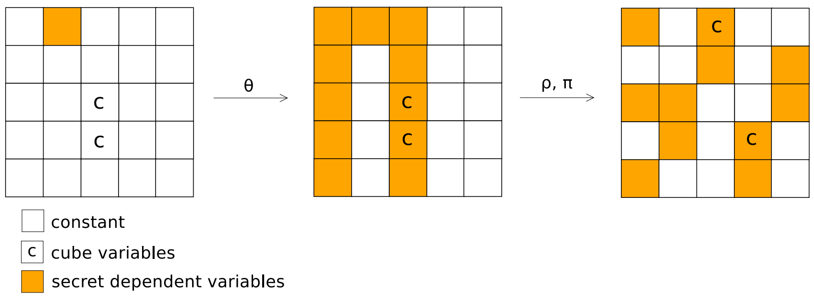

Fig. 9: Diffusion of the secret dependent variables

After the application of the linear mappings, we apply χ to the state. Despite its non-linearity, the only non-linear operation of χ is multiplying together bits in consecutive lanes (with the same y and z indexes). Thus, the bits of the 2 lanes that depend on the public variables, $A[2,0], A[3,3]$, are only multiplied with the bits of $A[1,0], A[3,0], A[2,3], A[4,3]$. Since these 4 lanes are constants (they do not depend on any cube or secret variables), then the cube variables are not multiplied by any variables throughout the first round (ι is a linear mapping which does not change this property). In other words, the symbolic form of each state bit $A[x,y,z]$ can be written as $L_{x,y,z}(v_1, v_2, \ldots, v_{32}, w_1, w_2, \ldots)$, where

$L_{x,y,z}$ is some linear function, and the variables w_i depend only on the secret variables (and not on the cube variables $v_1, v_2, \ldots, v_{32}$).

We now analyse the symbolic form of the state bits after 6 Keccak rounds, whose algebraic degree in the state variables after one round is $2^5 = 32$. Given the special symbolic form of the state bits after one Keccak round, the degree of each state bit after 6 rounds in the variables $v_1, v_2, \ldots, v_{32}, w_1, w_2, \ldots$ is at most 32, and thus the superpoly of the monomial $v_1 v_2 \ldots v_{32}$ is constant. As a result, the cube sum of each state bit after 6 rounds is a constant, which does not depend on the value of the secret variables. This proves Property 1.

Property 2 (restated). *The cube sums of the output bits after 6 rounds depend on the value of $A[0,0]$.*

Proof. When considering the bits of $A[0,0]$ as secret variables, they are multiplied with the cube variables in the first round (e.g. $A[0,0]$ diffuses to $A[1,0]$, whose bits are multiplied with $A[2,0]$ due to χ). After 6 rounds, we expect the degree of the output bits, in the state bits after one round, to be $2^5 = 32$. Consequently, we expect the superpoly of $v_1 v_2 \ldots v_{32}$ for an output bit to generally depend on the value of $A[0,0]$, and thus the cube sums of the output bits generally depend on the value of $A[0,0]$. This proves Property 2.

Simulation Results

In our 6- and 7-round key recovery attack, the most desired situation is when each 64-bit key would correspond to a distinct vector of cube sums. However, checking all 2^{64} cases, where each case requires summing over 2^{32} messages, is infeasible. Therefore, we conducted experiments checking a limited number of keys and using smaller cubes. First, we checked 2^{16} randomly chosen keys using 16-bit cube and nearly all keys have their unique cube sum vector. Only a very small fraction (below 0.007%) share the output vector with other keys. The second experiment, ran on the smaller variant with 400-bit state, with 2^{16} randomly chosen keys, showed that each key has its unique cube sum vector. Thus, our simulation results is a strong indication that assumptions taken for the 6- and 7-round key recovery attacks are sound.