Applications of Key Recovery Cube-attack-like*

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

[1] Institute of Computer Science, Polish Academy of Sciences, Poland
[2] Section of Informatics, University of Commerce, Kielce, Poland
[3] Queensland University of Technology, Brisbane, Australia

Abstract. In this paper, we describe a variant of the cube attack with much better-understood Preprocessing Phase, where complexity can be calculated without running the actual experiments and random-like search for the cubes. We apply our method to a few different cryptographic algorithms, showing that the method can be used against a wide range of cryptographic primitives, including hash functions and authenticated encryption schemes. We also show that our key-recovery approach could be a framework for side-channel attacks, where the attacker has to deal with random errors in measurements.

Keywords: Cryptanalysis, Cube Attacks, Hash Functions, Authenticated Encryption, Side-channel Attacks

1 Introduction

An approach to attack cryptographic algorithms is to take advantage of their potentially low algebraic degrees. A technique called higher-order differential cryptanalysis [16] was arguably the first one to exploit a low algebraic degree in block ciphers. In 2009, Dinur and Shamir formally introduced the cube attack — the key-recovery attack which can be seen as an extension of higher-order differential cryptanalysis and AIDA [22]. Since its introduction, the cube attack was applied to many different cryptographic primitives - see [4, 5, 17].

In [11], Dinur et al. attacked Keccak hash function with a variant of the cube attack with much better-understood Preprocessing Phase, where complexity can be calculated without running the actual experiments and random-like search for the cubes. In this paper, we go one step further and show that our method is not limited to a single algorithm (Keccak) but can be applied to analyse hash functions and authenticated ciphers as well as to side-channel attacks. We highlight a generic character of our approach and apply it in several different scenarios.

The main idea in our approach is to select the cube variables in such a way that the calculated sums depend only on a (relatively) small number of key bits, whose values can be recovered independently from 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 applied in the attacks on the stream ciphers Trivium and Grain in [14]. 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 a given algorithm. This analysis enables us to estimate the complexity of the attack beyond the feasible region (in contrast to the simulation-based attack of [14]).

First, as the point of reference, we describe our attack on Keccak working in the MAC mode. Keccak is a new cryptographic hash function adopted as the standard SHA-3. Then, we analyse a few authenticated encryption schemes, namely Keyak, HANUMAN, PRØST-APE. Recently, the crypto community focuses its attention on dedicated authentication encryption schemes, which aim to provide message encryption and authentication more efficiently. An interest in new efficient and secure solutions is manifested in the recently launched competition called CAESAR [1]. Keyak, HANUMAN and PRØST-APE were all submitted to the competition.

* This paper is an extension of our work published at Eurocrypt 2015[11]

Lastly, we show that our key-recovery approach could be a framework for side-channel attacks, where the attacker has to deal with random errors in measurements. Table 1 summarizes our results.

Table 1. Our key-recovery attacks. (First three results are taken from our Eurocrypt'15 paper [11], others are new.)

Algorithm	Rounds	Key size	Time	Data	Reference
Keccak-MAC	6	128	2^{66}	2^{64}	Sect. 4.2
Keccak-MAC	7	128	2^{97}	2^{64}	Sect. 4.3
Keyak	7	128	2^{76}	2^{75}	Sect. 4.4
PRØST-APE	5	256	2^{135}	2^{135}	Sect. 5
HANUMAN	6	120	2^{65}	2^{65}	Sect. 6
Keccak-MAC	3*	128	2^{26}	2^{26}	Sect. 7.1

*side-channel attack with the leakage after the 3rd round.

2 Cube Attack

The cube attack is a chosen plaintext key-recovery attack, formally introduced in [12]. Below we give a brief description of the cube attack, and refer the reader to [12] 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 dimension $d-1$ 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 [12]. 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 $v = (v_1, \dots, v_p)$ 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

$$L(x) = a_1 x_1 + \ldots + a_n x_n + c.$$

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

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

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

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 the Gaussian elimination.[4]

3 Overview of Divide-and-Conquer Cube Attack

Let us first describe the main steps of the attack with a simple example. We attack the MAC algorithm and the aim is to recover 128-bit key. An attacker can choose message bits and form a desired cube, also output bits (128-bit tag) are available. Let us assume that the attacker identifies the 32-bit cube such that the corresponding superpoly depends only on 16 key bits $k_0 \ldots k_{15}$. In Preprocessing, we create a table, whose rows are indexed by all 2^{16} possible values of the key $k_0, \ldots, k_15$. The table has 128 columns corresponding to 128 output bits and each entry contains a superpoly. Message bits which are not in the cube are set to an arbitrary constant, e.g., all zeroes. As we have 128 output bits (128-bit tag), the superpolys can be calculated with

Table 2. Table constructed in Preprocessing Phase

$k_0 \dots k_{15}$	superpoly evaluations
0000000000000000	0111010100111001 ... 0111100101
0000000000000001	1101101010110001 ... 1101000100
0000000000000010	1110010101010010 ... 0010110110
...	...
...	...
1111111111111111	0010001010101010 ... 0101010101

the reference to 128 polynomials. Thus, each combination of $k_0 \dots k_{15}$ corresponds to the 128-bit vector, as shown in Table 2.

The time complexity of Preprocessing is $2^{16} \cdot 2^{32} = 2^{48}$ and we need to store in memory 2^{16} rows of the table.

In Online Phase the key is unknown but certainly $k_0 \dots k_{15}$ take one of the possible combinations. The attacker sums over the cube chosen in Preprocessing and obtains the vector of superpolys evaluations. Then, $k_0 \dots k_{15}$ can be identified from the precomputed table. The time complexity of Online Phase is 2^{32}.

As we will show in the attacks on real algorithms, once the desired cube is identified, it is easy to find others for which the corresponding superpoly depends on subset of different key bits. Therefore, the whole key can be recovered in a divide-and-conquer manner.

This key recovery technique works provided that two conditions are fulfilled. First, the attacker has to know which key bits are in the superpoly for a chosen cube. Please note that, unlike in the standard cube attack, a superpoly does not have to be linear. The second condition is that each combination of key bits from the precomputed table should correspond to a unique vector of superpolys evaluations (cube sums). Otherwise a match in Online Phase would not be possible. Actually, the second condition can be slightly relaxed. Even if there are two (or more) rows with the same superpolys, we can still recover a subset of key bits that are the same for both rows and later eliminate the false positives with some trial calls to the algorithm.

In the toy example above, we simply assume that the attacker identifies the 32-bit cube such that the corresponding superpoly depends only on 16 key bits, without giving any details on methodology for finding such cubes. Most modern symmetric crypto algorithms have an iterative structure — a sequence of (nearly) identical rounds. In our attacks, we typically examine the first round, particularly an interaction between cube and key variables. Then, we try to select cube variables such that they are not multiplied together in the first round, but are multiplied with a (relatively) small number of key bits.

In next sections we show the attacks on real cryptographic (round-reduced) algorithms. First, we apply our method to the keyed hash function Keccak working in the MAC mode. Then, we attack three different authenticated encryption schemes. A common methodology should be a convincing argument that our divide-and-conquer approach can be applied to many cryptographic primitives, particularly to the SP networks.

4 Attacks on Keccak-MAC and Keyak

4.1 Description of Keccak

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

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

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.

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-dependent 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. 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.

The Keccak sponge function can be used in keyed mode, providing several different functionalities.

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 attacker 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 attacker to forge a MAC, namely, provide a valid message-tag pair (M, T) for a message M that has not been previously authenticated.

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. [6]. 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.

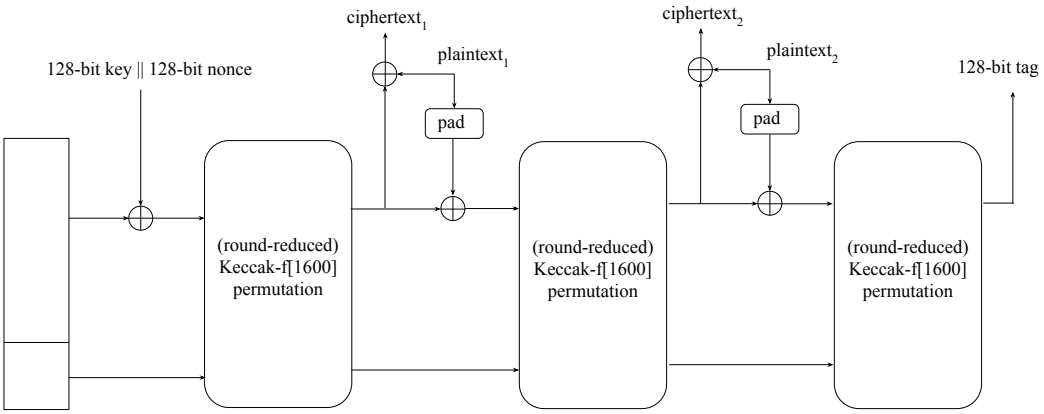

Fig. 2. Lake Keyak processing two plaintext blocks.

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 2 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].

Figure 2 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 (according to the Keyak specification).

4.2 Key-recovery Attack on 6-round Keccak-MAC

We attack the default variant of Keccak, suggested by the Keccak designers, with 1600-bit state and 1024-bit bitrate. The key and tag size are 128 bits. 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. 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 [11].

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.[5]
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.

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

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) 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}.

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 3 shows an impact of the auxiliary variables on diffusion of the secret key variables.

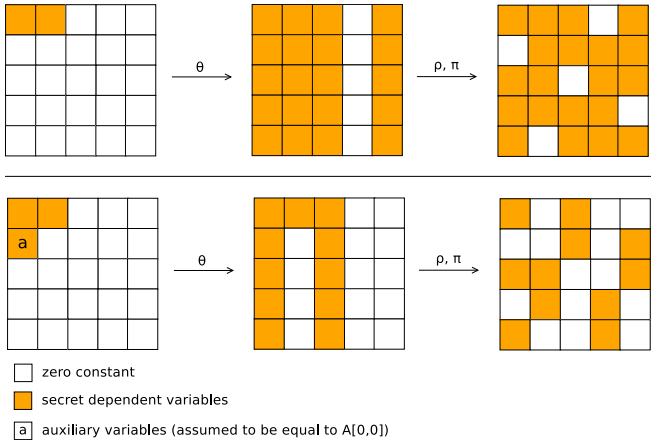

Fig. 3. Impact of auxiliary variables on diffusion of the secret key variables. When using auxiliary variables, $A[0,0]$ (secret variables) and $A[0,1]$ (auxiliary variables) are diffused to $A[0,0]$ and $A[1,3]$, without affecting many lanes of the state.

Similarly to the basic attack, we select the 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 [11], 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.

4.3 Key-recovery Attack on 7-round Keccak

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 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 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}.

4.4 Key-recovery Attack on 7-Round Keyak Authenticated Cipher

We now apply the divide-and-conquer attack to the 7-round Lake Keyak v1. We aim to break the authenticity and integrity of Keyak and in such a scenario we are allowed to reuse the nonce, according to the Keyak specification. 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. The forgery attack is done through the secret key recovery, after which the security of the system is completely compromised, that is, one can immediately forge the tag of any message.

For Lake Keyak v1, an input to the first permutation call is only the key and 128-bit nonce. Therefore, there is no such freedom of choosing cube variables from the state as we could see in the previous sections. To circumvent this limitation, we attack the algorithm in a slightly different way. We aim to recover the 1600-bit state obtained after the first permutation call. 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 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 property of χ. 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. Thus the algebraic degree is not affected by the last χ and we can attack 7 rounds using a 32-bit 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. With the selected 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 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 4 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 store the 2^{80} cube sums for all the possible values of the secret expressions in

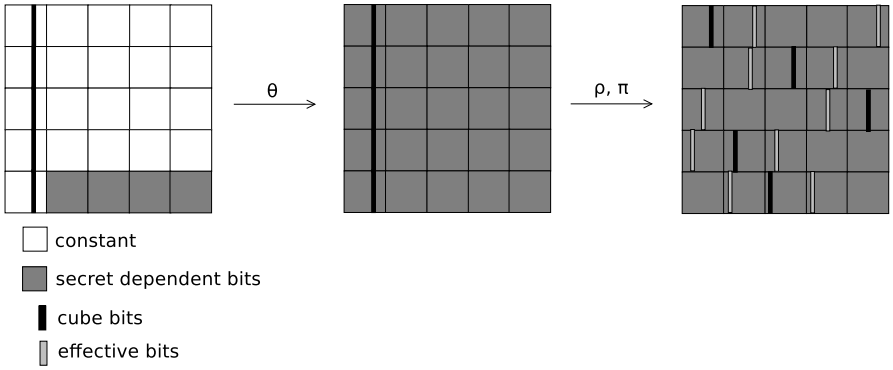

Fig. 4. Example placement of the cube and effective bits before the first χ is applied.

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 on 7-round Keyak 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.

5 Attack on PRØST-APE

5.1 Description of PRØST-APE Authenticated Cipher

PRØST-APE is an authenticated cipher introduced by Kavun et al. [15]. It is a permutation based scheme working in the APE mode [3]. PRØST-APE-$n[r, c]$ works on $2n$-bit state, with r-bit block size and a single c-bit key. In our analysis we focus on PRØST-APE-256[256, 256].

The first call to the underlying permutation has the nonce and the key as input. Once the initialization is done, the actual encryption starts, as shown in Figure 5.

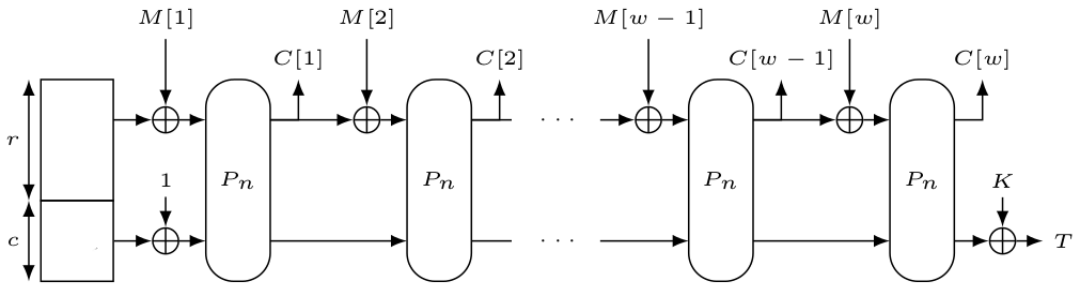

Fig. 5. Encrypting and authenticating w message blocks with PRØST-APE

The state of PRØST-256 can be seen a three-dimensional block $4 \times 4 \times 32$. The terms describing the parts of the state are borrowed from Keccak [8]. The PRØST-256 permutation has 18 (nearly identical) rounds and each round updates the state by means of the sequence of transformations: SubRows, MixSlices, ShiftPlanes, AddConstants. The SubRows step is a permutation consisting of the 4-bit Sbox applied to each row of the state. It is the only non-linear operation and the algebraic degree of the Sbox is 3. The MixSlices step uses the MDS matrix to mix bits in a given slice. Then, ShiftPlanes rotates the lanes in a given plane by a fixed offset. In AddConstants, different round/lane-dependent constants are XORed to each lane.

5.2 Attack on 5-round PRØST-256[256, 256]

The APE mode is the permutation-based AE scheme which offers nonce-misuse resistance. In our attack nonces are reused and we show that for the 5-round PRØST-256[256, 256], we are able to recover the key.

Let us notice that for PRØST-APE, before any ciphertext bit is available, the permutation has to be called (at least) two times. (The first call is for processing the nonce and key, and the second call for processing the first message block.) Consequently, this makes the attack harder as more rounds have to be overcome. However, instead of recovering the key directly, we can try to recover the capacity part of the state. So, we treat the capacity part as secret variables, message bits $M[2]$ (see Figure 5) as public, tweakable variables and ciphertext $C[2]$ as the output. Once the capacity part of the state is recovered, we run the permutation backwards and get the key.

In the attack, we use the 27-bit cube and as the algebraic of a round is 3, the cube 'covers' 3 rounds ($3^3 = 27$). Each cube variable is placed in a separate slice of the state. (For PRØST-256 there are 32 slices). The first SubRows step can be passed 'for free'. We can manipulate the message bits in such a way that the actual cube is formed after the first Sbox layer. Then, MixSlices mixes the bits in the slices, each cube bit interacts (bitwise XOR operation) with 8 capacity bits from its slice. Thus, potentially $8 \cdot 27 = 216$ capacity (secret) bits can be multiplied with the cube bits in the subsequent SubRows. In other words, the superpoly corresponding to the selected cube consists of (potentially) 216 capacity variables. This would lead to the precomputation table of 2^{216} rows, with a time cost of generating the table 2^{216+27}.

To lower the total complexity of the attack, we can use a birthday paradox argument, as we did for Keyak in Section 4.4. In Preprocessing, we calculate the cube sums for 2^{108} capacity bit combinations (instead of all 2^{216}). In Online Phase, we sum over the cube and repeat the procedure 2^{108} times, with a different nonce (or a different message block $M[1]$) for each summation. According to the birthday paradox, with high probability, the (unknown beforehand) values of the 216 capacity bits, in one of these online trials, will match one of their 2^{108} preprocessed values. This match will be detected by equating the cube sums, and allow us to recover the 216 capacity bits. Then, we guess the remaining 40 unknown capacity bits and are able to run the permutation backwards to get the key.

Thus, the memory complexity of the attack is 2^{108}, time complexity is $2 \cdot 2^{108} = 2^{109}$, and data complexity is $2^{108} \cdot 2^{27} = 2^{135}$ chosen plaintexts needed in Online Phase. Clearly, this is a theoretical attack but the complexity is much lower than the exhaustive search 2^{256}.

6 Attack on HANUMAN

6.1 Description of HANUMAN Authenticated Cipher

HANUMAN belongs to the authenticated encryption family PRIMATEs, introduced in [2] as a candidate for the CAESAR competition [1]. There are two variants of the scheme, with 80- and 120-bit security level. In our analysis we focus on HANUMAN-120. HANUMAN follows a design similar to that of SpongeWrap [7]. Figure 6 shows the scheme. At the heart of the scheme there are two permutations p_1 and p_4 which are nearly the same. (The only difference is values of round constants.)

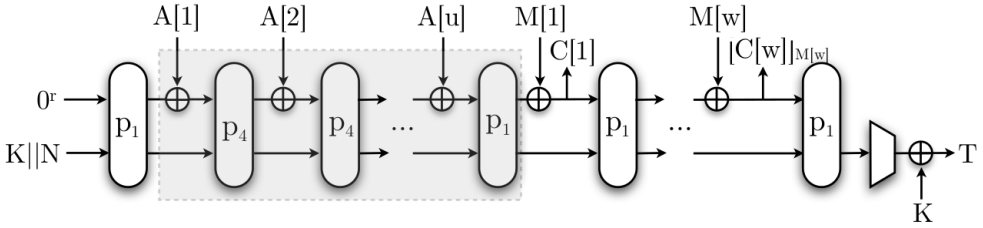

Fig. 6. The scheme of HANUMAN

HANUMAN-120 operates on the 280-bit state and the state is organized as the 7×8 array of 5-bit elements. The first row of the state ($8 \cdot 5 = 40$ bits) is the rate of the state whereas the rest of the state (240 bits) is the capacity. Before the first call of p_1, the 120-bit key K and 120-bit nonce N are XORed into the capacity part of the state.

Let us now specify the underlying permutation. We do not describe all the details but key features, essential for our attack. The permutation is designed according to the wide trail strategy and its structure is very similar to the AES cipher [10]. Both p_1 and p_4 permutations have 12 rounds and each round updates the state by means of the sequence of transformations: SubBytes (SB), ShiftRows (SR), MixColumns (MC), ConstantAddition (CA).

The SubBytes step is is a permutation consisting of the 5-bit Sbox applied to each element of the state. The algebraic degree of the Sbox is 2. The ShiftRows step is a byte transposition that shifts the rows over different offsets. The MixColumn is a linear operation, operating on column by column basis. Finally, the ConstantAddition step combines the second element of the second row with a predefined constant by a bitwise XOR operation.

6.2 Key Recovery Attack on 6-round HANUMAN-120

We attack the round-reduced variant of HANUMAN-120, where the p_1 permutation is reduced to 6 rounds and the other algorithm parameters remain the same as in the original specification.

For this attack, the selection method for the cube bits is slightly different that the one applied in previous sections. To break more rounds, we analyse interactions between key and nonce (cube) bits through 2 rounds. In the attack we use 61-bit cube; Figure 7 shows the placement of cube bits in the state.

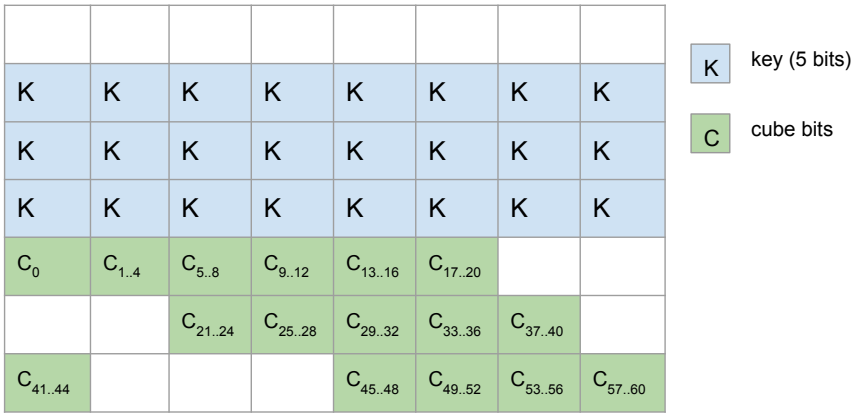

Fig. 7. Placement of key and cube bits in the state

Now we have to show that a superpoly corresponding to the selected 61-bit cube depends on only some part of the key — a requirement for our divide-and-conquer attack. First, let us argue that the cube variable c_0 is not multiplied with any other cube variables $c_1 \dots c_{60}$ in the first two rounds. Clearly, the first SBox cannot multiply out c_0 with any other variables, as c_0 is the only input variable to its Sbox. Then, ShiftRows shifts the bits in such a way that c_0 remains as the only cube variable in its column. So, through MixColumn c_0 interacts only with (15) key variables in its column and consequently the next SubBytes step does not multiply c_0 with any other cube variables.

Let us denote 280 state variables after two rounds by $t_0 \dots t_{279}$. Since the algebraic degree of a round is 2, then a degree of the output in $t_0 \dots t_{279}$ is, at most, $2^4 = 16$. To 'form' our cube $c_0 c_1 \cdot \dots \cdot c_{60}$ from the 16-variable monomial in $t_0 \dots t_{279}$, clearly we need to 'spend' one t variable

on c_0. Then, we are left with 15 t variables, where each such variable must be expressed by four cube variable ($15 \cdot 4 = 60$). If any t variable would be substituted by, say, monomial of three cube and one key variable, we would not be able to form our 61-bit cube, some cube variables would be missing. Therefore, we conclude that superpoly corresponding to the $c_0c_1 \cdot \ldots \cdot c_{60}$ cube depends only on those 15 key bits which are multiplied with c_0 in the second round.

Time complexity of the Preprocessing is $2^{61} \cdot 2^{15} = 2^{76}$ and for Online Phase is 2^{61}. It might be the case that the standard cube attack would be faster, but without implementing and running the Preprocessing, such conclusion is premature. As already stated, our divide-and-conquer attack is much better defined and much easier for precise complexity calculation.

7 Framework for Error-Tolerant Side-channel Attacks

The cube attack can be an effective tool for side-channel attacks.. In such the attacks, an attacker can learn some information about the intermediate variables (e.g., state registers). It is likely that the polynomials in the early rounds are of relatively low degree and hence a cube summation would be cheap. In [13], Dinur and Shamir showed how such the side-channel cube attack can be mounted. However, they use a strong assumption on measurement errors, namely the exact knowledge of error positions is known to the attacker and at least part of the measurements are error-free. Such the assumption is not what we usually see in practice, where each measurement is susceptible to some level of noise.

In this section we propose a framework for error-tolerant side-channel attacks, using our divide-and-conquer approach. Our framework handles errors in each measurement, bringing the model much closer to practice. In [18], such model is considered and the problem is converted to decoding a linear code. Our approach is different and seems conceptually simpler. Now, let us give the overview of the attack and then simulate a full key recovery on Keccak working in the MAC mode.

The attack works basically the same as the one described in Section 3. We assume that some bits leak after a number of rounds, so it becomes as if we attacked the round-reduced variants of a given cryptographic algorithm. Yet, there are some subtleties to be discussed.

We assume that a measurement of the leaked bit has an error $q < 1/2$. So, the probability that we get an accurate measurement is $1 - q$ and let us express it as $1/2 + \mu$. With $\mu = 0$, we would get a random guess. Each measurement can be treated as an independent event, so the probability that we get the accurate value of a cube summation can be obtained from the piling-up lemma [19].

$$Pr = 1/2 + 2^{t-1}\mu^t, \tag{1}$$

where t is a number of elements in a cube summation and the term $2^{t-1}\mu^t$ is called the bias ϵ.

As described in Section 3, we recover the unknown key by looking up to a precomputed table. The correspondence between the cube summations and key bits gives us a hint about the key bits. Here, however, the cube summations have some error probability, so we need to repeat the measurements to eventually detect the right key. Note that there is a good analogy to linear cryptanalysis, where usually only a fraction of plaintext-ciphertext pairs follow the linear characteristics and the attacker must observe a sufficient number of plaintext-ciphertext pairs to detect a 'signal' from 'noise'. Typically, a sufficient number of data is proportional to ϵ^{-2} [19].

In the side-channel attacks we cannot assume that the whole state or most of the state bits leak. More realistic approach is that only a small part of the state is available to the attacker. It brings us a problem that in the precomputed table many keys would share the same cube summation vectors (because a number of available output bits is small). However, usually, we can extend these vectors in a simple way. With public variables (e.g., message bits, IV bits,

nonce bits) fixed to different values, it is very likely that the cube summations will be different for a given key. Then, we just concatenate these vectors, obtaining a longer one and approaching to one-to-one correspondence between a key and cube summations — a desired situation for our attack.

7.1 Side-channel Key-recovery Attack on Keccak-MAC

We attack the Keccak hash function with the 800-bit state, working in the MAC mode. The key length is 80 bits and the capacity parameter set to 160 bits. Thus, the claimed security level is 80 bits.

We assume that the algorithm is implemented on a 8-bit processor. The attacker exploits the Hamming weight leakage when the state variables are loaded from the memory to the ALU. Such a scenario was used in a number of previous works, for example, in [20, 21]. In our attack, the leakage is after the 3rd round. We assume that the attacker can measure only 8 bytes of the state (their Hamming weights, to be precise) and in the attack we use only the least significant bits of these weights.

Preprocessing Phase As the leakage is after the 3rd round, the algebraic degree of the polynomials describing output bits is 2^3. Please note that calculation of the LSB of the Hamming weight of a given byte does not change the algebraic degree as it is a linear function.

The size of the cube variables is 4 and they are placed in such a way that the column parities remain constant. The reasoning behind the size and positions of cube variables is the same as the one used in attacks in Section 4.

It turns out that with the 4 chosen cube variables, there are typically 16 key bits which are multiplied with these cube variables in the first round. Therefore, the cube sums depends on the 16 key bits. Changing the position of the cube variables, we get different set of corresponding key bits, thus, we can mount the divide-and-conquer attack. The exact positions of cube variables and key bits in the corresponding superpolys are given in Appendix.

A cost of constructing the table is $2^{16} \cdot 2^4 = 2^{20}$. To make the cube summation vectors longer, we construct 10 separate tables, each table with different, randomly chosen message bits. This way we ensure that each key has a distinct (or nearly distinct) cube summation vector.

A single precomputed table helps us to recover 16 key bits. To recover more key bits, we construct 4 other tables, each one involving a different set of 16 key bits. Thus, the total cost of preprocessing is $2^{20} \cdot 10 \cdot 5 \cong 2^{26}$.

Online Phase In the online phase, we sum over the 4-bit cubes selected in the Preprocessing Phase. Having calculated the cube sums, we can look up into the tables to get the suggestion for the key bits. As stated before, our attack is error-tolerant and we assume that each measurement of the output bit could be incorrect with probability $q = 2^{-4}$. (The same value was used in one of the attacks from [18].) So, $1 - q = \frac{15}{16} = \frac{1}{2} + \frac{7}{16}$. The probability that we get the accurate value of a cube summation can be obtained from the piling-up lemma.

$$Pr = \frac{1}{2} + 2^{16-1} \cdot (\frac{7}{16})^{16} \cong \frac{1}{2} + \frac{1}{17} \tag{2}$$

So, the bias $\epsilon = \frac{1}{17}$ and a number of data (cube sums evaluations) needed for detecting the correct key should be proportional to ϵ^{-2}. Our experiments showed that measuring and calculating 3000 cube sums was enough to have the correct suggestion for a key in 97.8% cases. (Experiments done on 500 randomly chosen keys.) In those very few cases, where the correct key was not the first suggestion, it was very close to the first suggestion in the list.

The 80-bit secret key is recovered in a divide-and-conquer manner, using 5 different cubes. Each cube corresponds to a different superypoly, hence different key bits are recovered. (See Appendix for details.) As we sum over 4-bit cubes, the cost of cube summations is $5 \cdot 2^4 \cdot 3000 \cong 2^{18}$.

The overall time and memory complexity is dominated by Preprocessing and equals 2^{26}.

8 Conclusion

We showed that our key-recovery cube-attack-like can be applied to a wide range of cryptographic algorithms, particularly to the keyed hash functions and permutation-based AE schemes. Each attack reported in this paper follows the same 'core' methodology, though there are some details and constraints specific to a given algorithm. We also showed that our key-recovery approach could be a framework for side-channel attacks, where the attacker has to deal with random errors in measurements. It brings the attack model closer to reality and does not require strong assumptions such as exact knowledge of error positions in measurements. An inherent limitation of the method is an algebraic degree of an algorithm we attack.

Acknowledgement

This project was financed by National Science Centre, Poland, project registration number DEC-2014/15/B/ST6/05130

References

1. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness, http://competitions.cr.yp.to/caesar.html
2. Andreeva, E., Bilgin, B., Bogdanov, A., Luykx, A., Mendel, F., Mennink, B., Mouha, N., Wang, Q., Yasuda, K.: PRIMATEs v1, Submission to CAESAR Competition (2014), http://primates.ae/
3. Andreeva, E., Bilgin, B., Bogdanov, A., Luykx, A., Mennink, B., Mouha, N., Yasuda, K.: APE: Authenticated Permutation-Based Encryption for Lightweight Cryptography. In: Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers. pp. 168–186 (2014)
4. 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)
5. 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)
6. Bellare, M., Canetti, R., Krawczyk, H.: Message Authentication Using Hash Functions: the HMAC Construction. CryptoBytes 2(1), 12–15 (1996)
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 v1, http://keyak.noekeon.org
10. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Information Security and Cryptography, Springer (2002)
11. Dinur, I., Morawiecki, P., Pieprzyk, J., Srebrny, M., Straus, M.: Cube Attacks and Cube-Attack-Like Cryptanalysis on the Round-Reduced Keccak Sponge Function. In: Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I. pp. 733–761 (2015)
12. Dinur, I., Shamir, A.: Cube Attacks on Tweakable Black Box Polynomials. In: EUROCRYPT. pp. 278–299 (2009)
13. Dinur, I., Shamir, A.: Side channel cube attacks on block ciphers. Cryptology ePrint Archive, Report 2009/127 (2009), http://eprint.iacr.org/2009/127
14. 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)

15. Kavun, E.B., Lauridsen, M.M., Leander, G., Rechberger, C., Schwabe, P., Yalçın, T.: Prøst. CAESAR Proposal (2014), http://proest.compute.dtu.dk
16. 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)
17. Lathrop, J.: Cube Attacks on Cryptographic Hash Functions. Master's thesis, Rochester Institute of Technology (2009)
18. Li, Z., Zhang, B., Fan, J., Verbauwhede, I.: A new model for error-tolerant side-channel cube attacks. Cryptology ePrint Archive, Report 2015/447 (2015), http://eprint.iacr.org/2015/447
19. Matsui, M.: Linear cryptanalysis method for DES cipher. In: EUROCRYPT. pp. 386–397 (1993)
20. Oren, Y., Kirschbaum, M., Popp, T., Wool, A.: Algebraic Side-Channel Analysis in the Presence of Errors. In: Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings. pp. 428–442 (2010)
21. Renauld, M., Standaert, F., Veyrat-Charvillon, N.: Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In: Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings. pp. 97–111 (2009)
22. Vielhaber, M.: Breaking ONE.FIVIUM by AIDA an Algebraic IV Differential Attack. Cryptology ePrint Archive, Report 2007/413 (2007)

Appendix

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.

Details of Side-channel Attack in Section 7.1

4-variable cubes are set in that way that the XOR sum in the columns is zero. It limits diffusion in the first θ step. (The same idea was used in other attacks on Keccak in previous sections.) The 80-bit secret key is recovered in a divide-and-conquer manner, using five different cubes. Each cube corresponds to a different superypoly, hence different key bits are recovered.

1st cube: bits $16-19$ in the lane [3,2] and [3,3]
recovered key bits: 0, 1, 2, 3, 22, 23, 24, 25, 35, 36, 37, 38, 58, 59, 60, 61

2nd cube: bits $20-23$ in the lane [3,2] and [3,3]
recovered key bits: 4, 5, 6, 7, 26, 27, 28, 29, 32, 33, 39, 40, 41, 42, 62, 63

3rd cube: bits $24-27$ in the lane [3,2] and [3,3]
recovered key bits: 0, 1, 8, 9, 10, 11, 30, 31, 34, 35, 36, 37, 43, 44, 45, 46

4th cube: bits $28-31$ in the lane [3,2] and [3,3]
recovered key bits: 2, 3, 4, 5, 12, 13, 14, 15, 38, 39, 40, 41, 47, 48, 49, 50

5th cube: bits $20 - 23$ in the lane [3,1] and [3,2]
recovered key bits: 3, 4, 5, 6, 27, 28, 29, 30, 45, 46, 47, 51, 52, 53, 54, 77, 78, 79

The remaining 22 key bits can be recovered by exhaustive search or one may try other cubes which would 'cover' the remaining key bits.