Università degli Studi di Milano

Facoltà di Scienze e Tecnologie
Corso di Laurea in: Informatica

Analysis, implementation and testing of Keccak, the winner of the SHA-3 competition

Studente:
Filippo CEFFA

Relatore:
Prof. Andrea VISCONTI

Anno Accademico 2014/2015

Contents

1 Introduction

Keccak ([**10**]) is the hash algorithm winner of the SHA-3 competition, called by NIST in 2007 and lasted five years ([**13**]). Keccak is actually a wide family of algorithms, which differ in complexity, input and output size, padding choice and more. There are multiple variants of the algorithm, but only a few of them have been selected by the NIST. These include four versions of Keccak with varying hash length, four additional ones introduced after the winning of the SHA-3 competition, which are just a refined version of the previous ones, and two Shake algorithms that provide a variable output length.

Since its release the algorithm has been subject to studies and attacks to test its actual security. Joan Daemen and Gilles Van Assche proposed a powerful tool to analyse the differential complexity of Keccak: the differential trails ([**3**]). A trail describes the evolution of a differential path along multiple rounds, and the lower its weight, the higher is the probability to predict the state after the execution of Keccak.

Dinur, Dunkelman and Shamir in [**2**] proposed a subset attack to the three-rounds version of Keccak-512. It is based on the birthday paradox, and exploits an internal characteristic, which is similar to a differential trail between parts of a single state.

In this work Keccak has been fully implemented in all of its standard variants, as well as the algorithms to generate the three-rounds differential trails ([**3**]). The subset attack proposed by Dinur, Dunkelman and Shamir required multiple memory optimizations to become practical on a desktop PC. We implemented it as a two-steps attack, which stores possible collisions in the first iteration and identifies the actual matches during the second iteration. An outputs reduction technique was also introduced to minimize the amount of memory used by the attack, while maintaining the same collision probability. The memory cost of the attack was reduced by over 100 times, and the first collision was found on a desktop PC in less then 6 hours.

The subset attack was also modified to suit other variants of Keccak, including two (closed) challenges present on the official Keccak website ([**11**]), and collisions were found for those as well, in very little time.

This work is structured as follows:

Section 2 describes Keccak, its core algorithm Keccak-f (2.1) and the sponge construction (2.2).

Section 3 reports the currently known attacks for multiple versions of Keccak, the differential trails generation (3.1) and the subset attack for Keccak-512 (3.2).

Section 4 describes the implementation of Keccak (4.1), of trails generation algorithms (4.2) and the attacks implemented for Keccak-512 and the two Keccak-160 variants (4.3).

Section 5 reports examples of trails (5.1) and practical collisions (5.2).

Finally Appendix 1 contains the source code of the three-rounds Keccak-512 attack, described in Section 3 and Section 4.

The source code of this work can be found at **[14]**.

2 Description of Keccak

A sponge function is an algorithm that can accept an input of any size, and produce an output of any size. Keccak is a family of sponge functions, and thanks to this property it can be used for multiple purposes, from hashing to pseudo-random number generation. These functions define *how* the algorithm is used, but the core of Keccak, named Keccak-*f*, stays the same across different sponge constructions, and it's responsible for all the cryptographic operations that grant the security of Keccak, as shown in Figure 1.

Section 2.1 and Section 2.2 describe Keccak-*f* and the sponge construction respectively. A more detailed description of the algorithm, written by the authors themselves, can be found in [**1**] and [**4**], and on the official Keccak website ([**10**]).

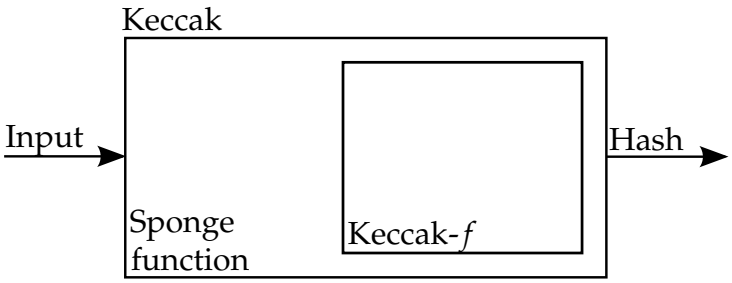

Figure 1: Keccak high-level view

2.1 Keccak-*f*

Keccak-*f* comes in 7 different variants, or permutations, whose main differences are the input size and the number of rounds.
The input is a three-dimensional matrix, where:

· $|x| = |y| = 5$

· $|z| = w = 2^l$, with l varying from 0 to 6.

It is useful to have a terminology to refer to subsets of this input matrix:

· Fixing no axis gives the whole matrix;

· Fixing x gives a sheet, y a plane and z a slice;

· Fixing z and y gives a row, x and z a column, x and y a lane;

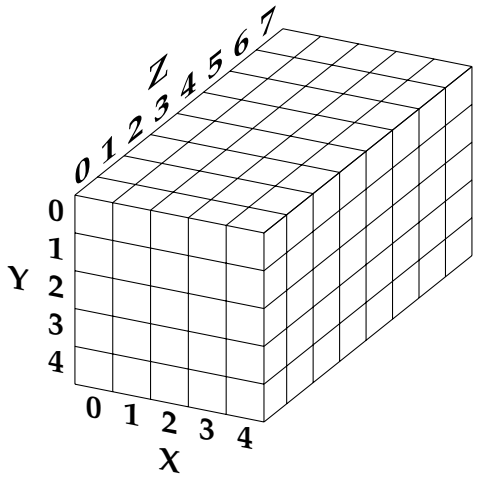

Figure 2: Keccak-f[200] input matrix with $l = 3$

· Fixing all the axes gives a single bit position;

Each version of Keccak-f is named Keccak-$f[b]$, where b is the number of input bits, given by the formula:

$$b = 25 * 2^l \tag{1}$$

Figure 2 represents the input matrix of Keccak-f[200], with l = 3, $w = 2^l = 8$ and $b = 25 * 8 = 200$.

Depending on the value of b, Keccak-f is iterated for a variable number of rounds:

$$n_r = 2 * l + 12 \tag{2}$$

All the variants of Keccak proposed to the SHA-3 competition use Keccak-f[1600] as their core algorithm, being it more secure and as easy to store in x64 architectures as the other permutations. All the Keccak-f variants have in common the structure of the round:

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta \tag{3}$$

These five functions are executed every round on the current state of the algorithm. θ, ρ and π are linear operations that provide diffusion, χ is a non-linear function, and therefore is the only operation that adds differential complexity to the algorithm, and finally ι simply adds a round constant to the first lane of the state. θ, ρ and π are often referred to as L, since they are the linear part of the round. Each of these functions will

be now presented more in depth, providing their mathematical formulas, while the implementation can be found in `Keccak_f_functions.c`.

From now on it is implied that $x, y \in Z_5$ and $z \in Z_w$.

2.1.1 Theta θ

Theta is the first linear function of the Keccak-f round, and when $n_r = 0$ it is directly applied to the input matrix. It is specifically built to increase the diffusion of the algorithm.

$$a[x][y][z] = a[x][y][z] + \sum_{y'=0}^{4} a[x+1][y'][z-1] + \sum_{y'=0}^{4} a[x-1][y'][z] \quad (4)$$

It is clear from the formula that theta can act as an identity operation if all the columns in the input matrix have an even number of active bits. This particular state is known as column parity kernel (CP-kernel) and it is fundamental in basically every attack in the bibliography.

2.1.2 Rho ρ

Rho is designed to provide inter-slice dispersion, and it does that by shifting each bit along the z axis, by a value different for each lane:

$$a[x][y][z] = a[x][y][z - (t+1)(t+2)/2] \quad (5)$$

t is generated by the following formula:

$$0 \leq t \leq 24 \text{ and } \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in GF}(5)^{2x2}$$

$$\text{or } t = -1 \text{ if } x = 0 \text{ and } y = 0 \quad (6)$$

Since those values are always the same across different Keccak instances, they can easily be precomputed and stored manually in a 5x5 matrix. Rho acts as an identity function in Keccak-f[25], because the lane length w is 1 and all the shift offsets are modulo w.

2.1.3 Pi π

Pi introduces dispersion by repositioning the bits inside of a slice. Each bit in position (x, y) is assigned to a new position (x', y') according to the following formula:

$$a[x][y][\cdot] = a[x'][y'][\cdot], \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \tag{7}$$

This is valid for every value of z, so π actually move the entire lanes to a new position.

2.1.4 Chi χ

Chi is the only non-linear step in Keccak-f. The value of each bit is:

$$a[x][\cdot][\cdot] = a[x][\cdot][\cdot] + (a[x+1][\cdot][\cdot] + 1) * a[x+2][\cdot][\cdot] \tag{8}$$

Since y and z are the same for each of the three bits involved in the function, each row of output is defined only by its correspondent row of input. Basically χ acts as an S-box on each row of the state, assigning an output given a certain input. Therefore the number of S-box is $5 * w$.

2.1.5 Iota ι

Iota is a very fast function. At each round it adds a round constant to the top-left lane ($x = 0$ and $y = 0$). The round constants RC are:

$$\text{RC}[i_r][0][0][2^j - 1] = rc[j + 7i_r] \text{ for } 0 \leq j \leq l \tag{9}$$

rc are the output values of a binary linear feedback shift register (LFSR):

$$rc[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x, \text{ in GF}(2)[x] \tag{10}$$

In `Keccak_f.c` we implemented the LFSR to generate these constants. However, since they are always the same, it is possible to save them inside an array and avoid to generate them every time. If $w < 64$, the constants are truncated to the correct number of bits.

Iota is built to break the symmetry of the state. A good example of its importance is the subset attack for the three rounds of Keccak-f[1600] in Section 3.2. The attack requires an high internal symmetry to be effective, and ι is the main reason that prevents to extend the attack of one extra round.

2.2 Sponge construction

Keccak needs to be able to handle different input sizes, and also to produce hashes with different lengths. To do so Keccak-f must be called multiple times, but, if the process is not done properly, the security of the algorithm could be compromised. The sponge construction is the solution to this problem adopted by Keccak, and it is defined by:

$$\text{Keccak}[r, c] \triangleq \text{Sponge}[\text{Keccak} - f[r + c], \text{pad}10^*1, r] \tag{11}$$

The number of input bits to Keccak-f is the bitrate r. The padding is c bit long, and it is built like this:

$$P = \text{ pad } 10^*1 \,||\, 1000...0001; \tag{12}$$

Since the number of input bits to Keccak-f is b, $r + c = b$.

From Definition 11 it is clear that the sponge function can be configured to use different permutations of Keccak-f, different padding choices and different bitrates. The structure of the sponge function remains the same though, and consists in two phases, an *absorbing phase* to process all the input, and a *squeezing phase*, to produce the output hash.

2.2.1 Absorbing phase

In the absorbing phase the input of Keccak is read and processed. The message M is padded so that $|\text{M}| \bmod r = 0$, and then it is divided in chunks of length r. One by one they are xored to the first r bits of the current state, and then fed to a new execution of Keccak-f, as shown in Figure 3. The iteration stops when all the blocks of the input message are read and hashed.

2.2.2 Squeezing phase

At the end of the absorbing phase an output of length $r + c$ is produced. But Keccak must be able to generate hashes of any length, and it does that during the squeezing phase. Keccak-f is iterated, and every time the first r bits of the output are added to the final hash, until the required length is reached, as shown in Figure 4.

Keccak 224, 256, 384, 512 and SHA 224, 256, 384 and 512, the standard versions of the algorithm, actually don't need the squeezing phase, because the length of the output generated by the absorbing phase is greater

than the required hash length.

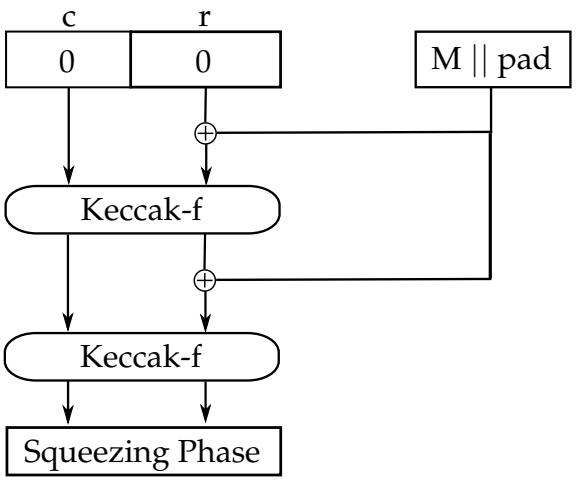

Figure 3: Keccak absorbing phase

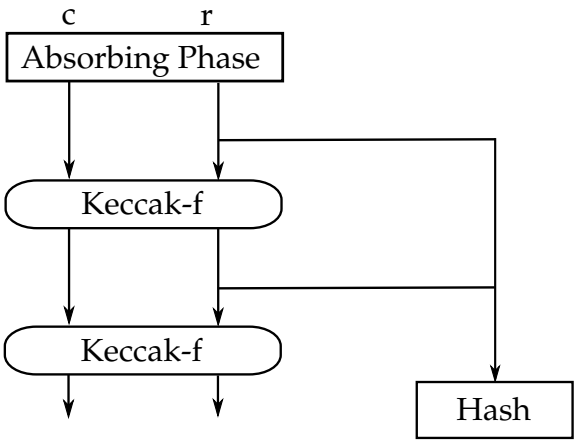

Figure 4: Keccak squeezing phase

3 Known attacks and weaknesses

Keccak is a very wide family of sponge functions, and as such there is a great number of possible attacks. Most of them exploit weaknesses in the sponge function, rather then attacking directly the core algorithm, Keccak-f. Others are directed to non-standard Keccak variants, to possible implementations of the algorithm or even to versions of Keccak not used for hashing ([**8**] and [**12**]). However, since our main interest was the security of Keccak-f, we chose to focus on the standard Keccak versions approved by NIST, because their sponge construction is very simple, and for short-length inputs they just consist in a single Keccak-f iteration.

There are three main kinds of attacks that can be applied to Keccak, and to any other hash algorithm as well:

· *Preimage attack*:
  given an output o, find an input i so that $o = h(i)$.

· *Second preimage attack*:
  given an input i, find an input i' so that $h(i) = h(i')$.

· *Collision attack*:
  find two inputs i and i' so that $h(i) = h(i')$.

Keccak is very resistant to first and second preimage attacks. Most of the work on these kinds of attacks has been done by Pawell Morawiecki ([**5**], [**6**] and [**9**]). Only two rounds have been practically attacked in preimage attacks, and none in second preimage attacks, however Morawiecki proposed non-practical attacks up to nine rounds that succeed in reducing the total complexity of the algorithm.

Version	Rounds
Keccak-224	4
Keccak-256	4
Keccak-384	3
Keccak-512	3

Figure 5: Number of rounds reached in collision attacks.

Figure 5 reports the number of rounds reached in practical collision attacks in the four standard Keccak variants. These results were possible

thanks to the work of Itai Dinur, Orr Dunkelman, and Adi Shamir, who described the attacks used to obtain them in [**2**] and [**7**].

The algorithm is still very secure at the time of this writing, since each of these four versions uses Keccak-f[1600] as its core function, whose number of rounds is 24. There are also attacks for a greater number of rounds, which are not-practical, but succeed in reducing the total security of the algorithm. None of them has ever reached the full 24 rounds Keccak-f[1600] though, and not even the 12 rounds employed by Keccak-f[25].

This work was mainly focused on collision attacks, and we manage to find actual collisions for Keccak-512 reduced to three rounds with the implementation of the attack described in [**2**].

There are also many studies that test the security of Keccak from a more theoretical point of view without attacking it directly, but showing potential weaknesses of the algorithm that could be exploited by future attacks. Such is the case of the work of Joan Daemen and Gilles Van Assche, who studied the differential weaknesses of Keccak in [**3**].

Section 3.1 describes the differential trails proposed in [**3**] more in depth.

Section 3.2 reports the subset attack proposed by Itai Dinur, Orr Dunkelman, and Adi Shamir in [**2**].

3.1 Differential trails

Differential cryptanalysis is a branch of cryptography that studies how differences in the input of a cypher can influence its output. If an algorithm has a predictable difference evolution, it can be subject to several attacks. DES, for example, was definitely broken and declared insecure due to chosen plaintext attacks, which exploit the differential weakness of the algorithm. To analyse the differential complexity of Keccak it is useful to calculate the lowest-weight differential trails. Joan Daemen and Gilles Van Assche wrote the most important paper on Keccak's trails ([**3**]), and most of the concepts described in this section are based on their work.

For the implementation of trails generation algorithms made in this work, refer to section 4.2.

3.1.1 Trail definition

A trail is a differential path across a certain number of rounds. The difference d and the number of rounds n must be predefined for each trail. $m1$ and $m2$ are two plaintexts with difference d, while the Keccak-f algorithm

is here denoted by h. p is the probability that the following expression is true:

$$h(m1) \oplus h(m2) = h(m1 \oplus m2) \tag{13}$$

Finally, the weight w of a trail is:

$$w = log_2\ p \tag{14}$$

A trail of difference d across n rounds has weight w if the probability of finding two messages $m1$ and $m2$ with difference d, which satisfy the formula, is $1/2^w$.

The resistance of the algorithm to differential attacks, reduced to n rounds, is directly connected to the lowest weight trail for n rounds. Therefore the goal is to find the difference(s) d that gives the lowest-weight to the trail(s) for n rounds.

3.1.2 Intro to trails generation

All Keccak-f functions, except for χ, are linear steps.
This imply the following:

$$s(m1) \oplus s(m2) = s(m1 \oplus m2) \tag{15}$$

This means that there is no difference between:

· Pass each input to s, and then get their difference;

· Pass the difference of the inputs to s;

Since this is always true for linear functions, the probability p of Equation 13 is 1 (Equation 13 and 15 are exactly the same), and the weight of the step is zero. Therefore no weight is ever added to the trails due to the linear steps, the only function that can reduce the probability is χ, because to a single difference input is associated an affine subspace of outputs.

In Figure 6, for each input difference are reported the bases of its output subspace, and an offset value. For example, for input 00011, the output values are all the solutions of the following formula, with different values of the a_x coefficients:

$$output = a_1 * 00010 + a_2 * 00100 + a_3 * 01000 + 00001$$

An output space with n bases can generate 2^n outputs (including cyclic shifts of the 5 bits). Each output subspace include a value identical to the input difference, for example, for an input of 11111:

	forward propagation							
Difference	offset	base elements				$w(\cdot)$	$w^{rev}(\cdot)$	$\|\| \cdot \|\|$
00000	00000					0	0	0
00001	00001	00010	00100			2	2	1
00011	00001	00010	00100	01000		3	2	2
00101	00001	00010	01100	10000		3	2	2
10101	00001	00010	01100	10001		3	3	3
00111	00001	00010	00100	01000	10000	4	2	3
01111	00001	00011	00100	01000	10000	4	3	4
11111	00001	00011	00110	01100	11000	4	3	5

Figure 6: χ affine subspace of output differeces, with respective weight, reverse weight and Hamming weight. Source: **[3]**.

$$11111 = 0 * 00011 + 1 * 00110 + 0 * 01100 + 1 * 11000 + 00001$$

This means that χ, for a space with n bases, has $1/2^n$ probability of acting as an identity function, and therefore its weight is n. When a trail reach χ, it treats it as an identity step, leaving the current difference untouched, but adding to the total weight the weight of each row.

Since 00000 has weight 0, it is important to reach the χ round with very few active bits, in order to minimize the weight of the trail. In order to achieve this, it is mandatory to also consider the linear steps in the trails generation, because, while they don't directly add weight, they can drastically modify the current difference state. If the new state has a lot of active bits, a lot of weight will be added when the χ step is reached, and the final trail will not be useful. The behaviour of the linear steps is the following:

- ι does not modify the difference in any way, so it can be ignored.
- ρ shifts the difference bits along the z axis, but does not modify their number.
- π switches the bit positions along the x and y axis, without adding extra active bits.
- θ is the only linear step that modifies the number of active bits (usually adding them, for an input with few active bits), and it does so if some of the state columns have an odd number of bits.
  If all of them have an even number of bits, the state is called CP-Kernel, and θ act as identity, which is the ideal case.

A few active bits and maintaining the CP-Kernel are the basis of all the techniques to generate trails described in the next section.

Before proceeding with the generation of trails, it is useful to define the propagation weight, reverse propagation weight and Hamming weight of a state:

- The propagation weight describes the probability to generate an output row given a certain input row, and it is just another name for the weight of a trail defined above.
- The reverse propagation weight describes the probability that an output row has been generated by a certain input row.
- The Hamming weight is the number of active bits in a row.

The only weight used for three-rounds trails generation is the propagation weight, however the Hamming weight is always a good way to measure the quality of a difference state, and the reverse propagation weight is used to extend a trail to more rounds with a process called backward trail extension ([3]). The values of the weights for each input row are reported in Figure 6.

3.1.3 Trails generation

Keccak-f executes the five round functions in the following order:

$$\theta \rightarrow \rho \rightarrow \pi \rightarrow \chi \rightarrow \iota \tag{16}$$

While building a trail, it is possible to remove the linear steps before χ in the first round, because they could eventually be reconstructed backwards. By calling λ the linear steps before χ and removing ι, it is possible to write the path of a trail as follows:

$$\chi_0 \rightarrow \lambda_1 \rightarrow \chi_1 \rightarrow \lambda_2 \rightarrow ... \rightarrow \lambda_n \rightarrow \chi_n \tag{17}$$

To generate the lowest-weight trails for one round, which consists of a single call to χ, it is possible to place one bit in any position. χ will add 2 to the weight, as evident from the table shown in Figure 6.

To generate the lowest-weight trail for two rounds, two active bits must be placed in a single column. χ will immediately add 4 to the weight (2+2). θ will pass as an identity function, because the bits were evenly placed in a single column, then ρ and π will move those bits to different positions, and finally χ will add weight again, 2 for each line with an active bit, 4 in total. The lowest-weight for a two-rounds trail is then 4+4 = 8.

Extending the trails to an extra round is quite problematic: it is not possible to use a single column of bits anymore, because at the end of the second round the state is no more in the CP-Kernel, and the lowest weight trail found in this way, by Duc et al., is of weight 4+4+24 = 32. To maintain the CP-Kernel for another round, vortices must be used. A vortex is basically a set of n columns (from 2 to 5) of two bits each. The second bit of each column must have the same y of the first bit of the next column, forming a sort of chain.

This is shown more clearly in Figure 7.

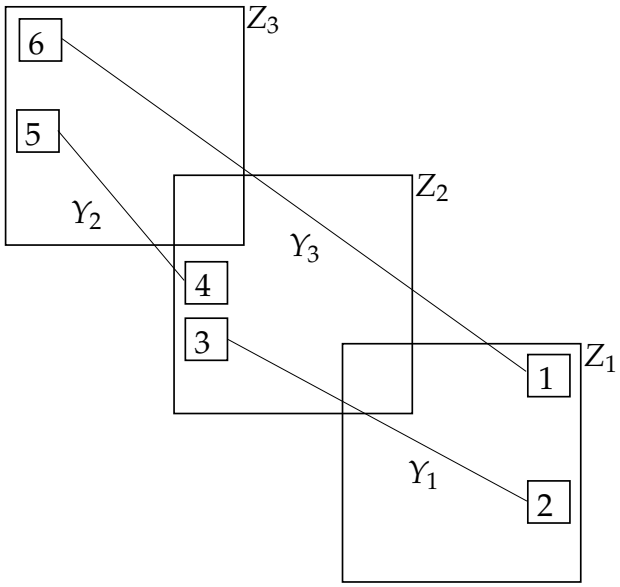

Figure 7: Structure of a 6-bits vortex

Having more than one column adds an interesting property to the trail: it is possible that, for some input differences, ρ and π will move the active bits to new positions, so that they will form new columns, maintaining the CP-Kernel. For a 6-bits vortex, the initial weight is 12 (2 multiplied by 6 bits), and, if the CP-Kernel is mainteined for all three rounds, a weight of $12 * 3 = 36$ is to be expected. It is indeed possible to obtain trails of weight 36 using 6-bits vortices. 4-bits vortices would be a better choice, because the weight of the trail, in the optimal case, would be $8 * 3 = 24$. However they are very few in number compared to the 6-bits vortices, and none of them leads to a trail with a weight lower than 36.

A trail of weight 36 is actually worse than the trail of weight 32 found with the naive method, but it is immediate to see that the trend of the trail generated with the vortices is more regular. For a number of rounds

greater than three, the second approach will be the only feasible option. Joan Daemen and Gilles Van Assche were able to find a trail of weight 74 for six rounds, which is just a bit more then 12 for round. This result would not be possible without the use of vortices.

The problem with differential trails is that it is not clear how to use them in a differential attack and I've not found practical attacks based directly on them. However the subset attack to Keccak-512, which will be described in Section 3.2, exploits an internal characteristic, which is similar to a trail, but takes into account differences between parts of a single input state.

3.2 Subset attack

This section reports the attack for Keccak-512 reduced to three rounds proposed in [**2**] by Dinur, Dunkelman and Shamir. First is described the general idea of the attack, then the choice of the characteristic and finally the time and memory complexity of the whole attack.

For the practical implementation of the attack refer to 4.3.

3.2.1 The general idea

The subset attack is based on the birthday paradox, which states that, given a space of values of size n, $n/2$ items must be considered to have a probability of 50% that two of them will have the same value. Therefore the subset attack goal is to reduce the space in which all the outputs of Keccak will surely be included. Then it is sufficient to try more than $\sqrt{n}$ inputs, and a collision should be found. As shown in Figure 8, the problem is that the inputs cannot be randomly chosen, or the output space will immediately grow so large, that even its root square would be a number too big to make the attack practically feasible. The solution to this is to carefully chose the inputs so that they will follow an internal characteristic, which is very similar to the trails explained in Section 3.1.

3.2.2 Internal characteristic

While in trails generation the difference is between two inputs, in a characteristic the difference is between groups of slices of the input matrix. The parameter i defines how many slices are in a single group.

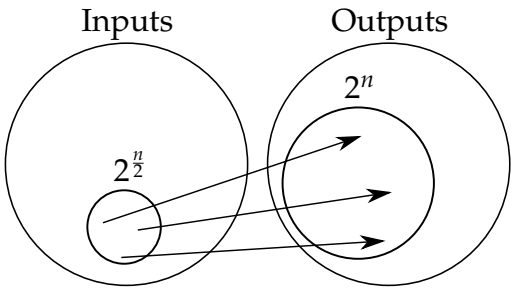

Figure 8: Inputs and outputs space in a subset attack

For example in Figure 9 there are 4 blocks of slices, so $i = 4$, and because $w = 16$, there are 4 blocks with 4 slices each. The difference is a three-dimensional matrix of the same size of the state, where each bit is 0 if the correspondent bit is equal to the other w/i ones, 1 otherwise. As in trails generation, a good initial difference must be found in order to lead to a low-weight characteristic. Since the differences are internal in a characteristic, a matrix full of zeroes is also a possible difference, and this happens when the blocks are identical. This particular case is shown in Figure 9 (right): the state repeats itself every i bits along the z axis, because the difference between the blocks is initially zero for each bit, so the characteristic has no active bits (left).

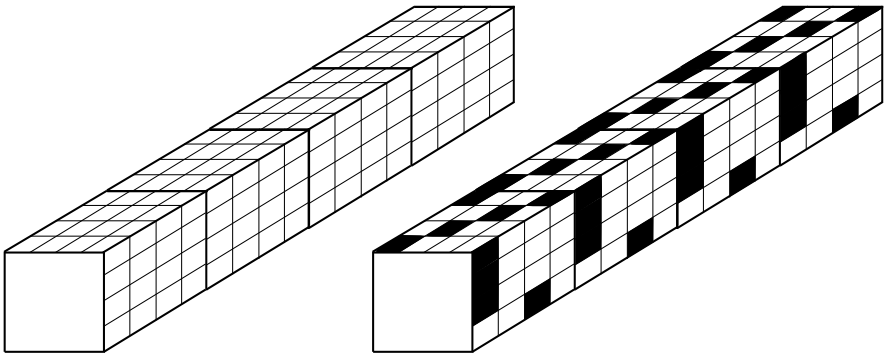

Figure 9: An example of input (right) with zero-characteristic (left)

The goal of this first part of the attack is to generate a trail 1.5 rounds long (stopping just before the χ of the second round). A longer trail would have a weight so high that the subset attack would not be practical.

The main reason why it is not possible to generate a characteristic as

long as the trails described in Section 3.1 is ι. This function was irrelevant in classic trails, but now the addition of a constant to the first lane introduces active bits in the differences between blocks, and makes it impossible to maintain the CP-Kernel as long as before.

Using the characteristic described above leads to 11 active bits after 1.5 rounds with probability equals to 1. This means that an input with that internal difference leads to an output space of size 1 after 1.5 rounds. We also tried to generate characteristics with other differences, for examples with a column of active bits as in classic trails generation, but the best choice seems to be the zero-characteristic described above by a wide margin. This characteristic is reported in Section 5.1.

3.2.3 Extending the characteristic

As explained in the previous section, the characteristic cannot go further than 1.5 rounds. This is immediately evident from the 11 active bits, which would lead to a very high weight for longer trails. From there, the first step is going over χ, which will add all the weight of the characteristic's final state. Since there are 11 active bits, in different rows, 22 weight is added, which means that the output space could contain 2^{22} different outputs.

Then it is possible to ignore the linear steps of the third round, but to pass the final χ the only thing to do is to consider all the possible values of the S-boxes that will generate the hash. Since the hash is 512 bit long, the number of lanes in which it will be contained is $512/64 = 8$. Therefore the number of S-boxes involved is $i \times 2$, because the hash is included in the first two rows ($5 \leq 8 < 10$), and the block has width i. Each one of these S-boxes has 5 bits as input and as output, so the number of possible values for each one of them is 2^5. Therefore the total number of possible inputs to the final χ, to represent a 512 bit hash is:

$$n_i = 2^{10 \times i} \tag{18}$$

Since each input to the χ_2 step is actually one of the 2^{22} possible values contained in the output space of χ_1, the number of inputs to χ_2 must be multiplied by that number. This is schematically represented by the following graph:

$$\lambda_0 \xrightarrow{2^0} \chi_0 \xrightarrow{2^0} \lambda_1 \xrightarrow{2^0} \chi_1 \xrightarrow{2^{22}} \lambda_2 \xrightarrow{2^{22}} \chi_2 \xrightarrow{2^{22+10\times i}} \tag{19}$$

and formalized by the following formula:

$$\text{size} = 2^{22} \times n_i = 2^{22+(10\times i)} \tag{20}$$

This is the final size of the subset, so after $\sqrt{\text{size}}$ inputs, due to the birthday paradox, a collision should be found.

Clearly a lower value of i leads to a smaller subset, making the attack faster, but a value of i too low may lead to a number of inputs not high enough to reach the square root of the output subset. Therefore the ideal value of i is the smaller one satisfying the following condition:

$$\text{number of inputs} > \sqrt{|\text{outputs subspace}|} \tag{21}$$

Keccak-512 has a bitrate r of 576 bits, so the number of lanes involved is $576/64 = 9$. Considering that the last lane has 2 bit of padding in it, then the number of inputs is:

$$2^{8*i+(i/2)} \tag{22}$$

The formula to find the correct value of i becomes:

$$2^{8*i+(i/2)} > \sqrt{2^{22+(10\times i)}} \tag{23}$$

The smaller value that satisfy the condition is $i = 4$, because the subset contains 2^{62} values, and 2^{34} inputs are sufficient to cover 2^{31} outputs.

With these values it is possible to proceed to the implementation of the attack, but before moving forward it is important to see how this kind of attacks behaves with the four-rounds Keccak-512 version. The subset attack is very good for the three-rounds Keccak-512, but adding a single round will introduce all the weight of each single S-box of the χ_2 step. This is more clear from the following scheme:

- $\lambda_0 \xrightarrow{2^0} \chi_0 \xrightarrow{2^0}$
- $\lambda_1 \xrightarrow{2^0} \chi_1 \xrightarrow{2^{22}}$
- $\lambda_2 \xrightarrow{2^{22}} \chi_2 \xrightarrow{2^{22+25\times i}}$
- $\lambda_3 \xrightarrow{2^{22+25\times i}} \chi_3 \xrightarrow{2^{22+(25+10)\times i}}$

This is not feasible for a reasonable number of inputs, the only alternative is to find a low weight characteristic for 2.5 rounds, and then move forward as usual:

- $\lambda_0 \xrightarrow{2^0} \chi_0 \xrightarrow{2^{?}1}$

- $\lambda_1 \xrightarrow{2^?1} \chi_1 \xrightarrow{2^?1+^?2}$
- $\lambda_2 \xrightarrow{2^?1+^?2} \chi_2 \xrightarrow{2^?1+^?2+^?3}$
- $\lambda_3 \xrightarrow{2^?1+^?2+^?3} \chi_3 \xrightarrow{2^?1+^?2+^?3+(10\times i)}$

Unfortunately trying to find a 2.5 rounds characteristic with a low weight is not an easy task, and we weren't even able to find one with a weight smaller than 100. This does not imply that these trails don't exists, but there is not a way to find them yet.

This concludes the theory behind the subset attack. Section 4.3 describes its implementation, and the strategies we adopted to make it practical on an average desktop PC.

4 This work: implementation, testing and optimization

This chapter analyses our work on Keccak.

Section 4.1 describes the implementation of Keccak-f and the standard variants of Keccak.

Section 4.2 reports the algorithms to generate the differential trails for Keccak-f.

Section 4.3 first reports in details the attack implemented for the three-rounds version of Keccak-512, and its memory optimizations, then describes the attacks to two Keccak-160 variants, based on the original subset attack described in [**2**].

4.1 Keccak implementation

Keccak was implemented entirely in the C language, on a notebook PC running Ubuntu 14.04.

The code for Keccak-f and for the round constants generation with the LFSR for ι is contained in `Keccak_f.c`, while the five rounds functions and the generation of shift and rotation values for π and ρ are implemented in `Keccak_f_functions.c`. The algorithm can be set up to behave like any of the Keccak-f[b] functions, with $b = 25 * 2^l$, $0 \leq l \leq 7$, even though only Keccak-f[1600] is used by the standard sponge functions.

The implementation of the sponge construction is inside `Sponge.c`, and it can be set up with various parameters, like r, c, b, the padding or the hash output length. The code is divided into two main functions, the absorbing phase and the squeezing phase.

Finally, inside `MainKeccak.c`, are implemented the algorithms for the four Keccak, the four SHA and the two Shake standard variants. The only difference between this functions are the setup parameters they pass to the sponge construction, while Keccak-f stays the same across all these versions.

A few parameters were also added to the final program, like the choice between ascii or binary input, or the length of the hash in the Shake variants.

We also made a GUI version of the program using Qt libraries, as shown in Figure 10

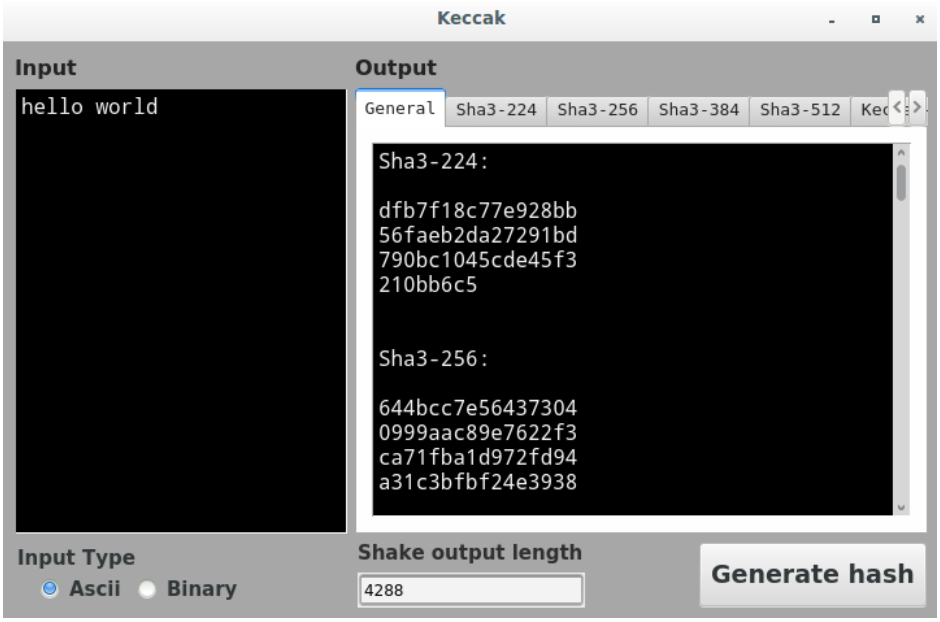

Figure 10: The GUI version of the Keccak implementation

4.2 Differential trails

In this work we implemented all the useful algorithms to generate three-rounds trails, as explained in section 3.1. The source code can be found in `Trails.c`.

The algorithm to generate trails with the method proposed by Duc et al. found 64 possible differences that lead to an optimal weight of 32. The trails generation algorithm that uses vortices found 178 different trails with a weight of 36. It can be easily adapted to use 8-bits vortices, and a few collisions with the optimal weight of 48 were found. As already stated in Section 3.1, no low-weight trails were found with 4-bits vortices.

The code also contains the implementation of a few useful functions to calculate the Hamming weight, propagation weight and reverse propagation weight of a state of Keccak.

4.3 Subset attack

In this section are described the practical attacks implemented in this work, based on the subset attack proposed in [**2**] by Dinur, Dunkelman and Shamir. First is described the naive version of the attack to Keccak-512,

then are explained all the memory optimization techniques we used to make the attack practical on a desktop PC. These include a two-steps variant of the attack to cache possible collisions and an output reduction technique to further minimize the memory cost. Finally are described the subset attacks adapted to two Keccak-160 variants.

In Appendix 1 is reported the source code of the practical three-rounds Keccak-512 attack described in this chapter.

4.3.1 Non-practical attack implementation

As described in Section 3.2, the values chosen for the attack are $i = 4$ and a zero-characteristic. The 2^{34} inputs are generated and Keccak-512 is run on each one of them. The input-output pair is then stored in memory. If another identical output is already present, a collision is found. We first tried to use red-black trees to handle memory, because the insert time has a best case scenario of $\Theta(1)$, compared to B-trees which always have an insert time of $\Theta(\log(n))$. In the end we used an hash table instead, because it always grants an insertion and search time of $\Theta(1)$, requires less memory and has no rebalancing cost. The time cost of hashing is irrelevant compared to its benefits.

The hash table used in the attack is called sparsehash, and the source code is in `libchash.c`. As its name suggests, it is a sparse hash table, which grants a negligible memory overhead, a couple of bits for each entry on average.

Even with this memory optimization the attack is far from practical on a desktop PC. The output of Keccak-512 is 512 bits long, while the input can be represented as a 34 bit index, instead of saving the whole matrix. Therefore the entry size must be at least:

$$512/8 + 34/8 \approx 69 \text{ B},$$

and the total RAM required amounts to about

$$2^{34} * 69/2^{30} = 1104 \text{ GB}.$$

Potentially a collision could be found after only 2^{31} memory entries, but the required RAM would still be ≈ 136 GB. On smaller variants of Keccak this attack could work without further optimizations, but for the three-rounds of Keccak-512 a two-steps attack was required to reduce memory usage. This is described in the next section.

4.3.2 Practical attack implementation

The number of hash table entries is the main reason that makes the attack heavily memory dependent. So we reduced the number of entries saved in memory by dividing the attack in two phases:

· In the first phase possible collisions are detected and saved in memory.

· In the second phase the actual collisions are found.

Detecting possible collisions during the first phase requires a cache array and a hash function with a fixed output size and a good diffusion.
The hash function used in the attack, xxHash, can be found in `xxhash.c`. It is very fast and provides a good diffusion, but any other hash function with these characteristics would work just as well.

At the beginning of the attack the array is preallocated, and its size is 2^n bits. Then each output of Keccak-512 is calculated, but instead of being stored in memory, it is hashed with the hash function to produce a n bits number. This number is used as an index to access the cache array: $n/8$ is the position inside the bytes array, n mod 8 is the bit offset inside the target byte. The target bit is then set to 1, but if it already has a value of 1 the current input-output pair is stored in the sparse hash table described in the previous section, with the output as key.

Hashing the output caused an information loss, and multiple outputs could be mapped to the same hash, so the collision is just a possible collision. It is important to note that there is no information left about the first of the two outputs that generated the same hash, so only the second input-output pair of each possible collision is saved to memory. At the end of this first phase, the memory will contain all these input-output pairs, and the cache array will no longer be useful, so it can be freed.

The second phase requires to calculate again all the outputs of Keccak-512 for each input. The outputs will be used to search the hash table. When a match is found, the inputs must be compared: if they are different an actual collision is found, because two different inputs generated the same output.

With this system the memory usage is drastically reduced, and the only drawback is the doubling of the time of execution of the attack. However, as described in the next section, the memory required will still be a little too much for an average desktop PC. Reducing the number of inputs is an option, but it will also lower the chances of finding a collision. The solution we adopted instead is to reduce the number of outputs. The differences between the two approaches is that reducing the number of inputs will lead to an output subspace of the same size as before, but with

less outputs stored in it. This reduces drastically the chances of finding a collision. However if all the inputs are passed to Keccak-512, and then the outputs are reduced, the subspace will be smaller but with the same density as before.

An example to better understand this: if all the outputs with a bit 0 in the first position are discarded, the final subspace will be halved, because it won't contain outputs with a leading 1. Doing the same thing on the inputs will just half the memory entries, but their output space will be exactly the same as before, because it is impossible to know which outputs will be discarded. It is important to notice that with this method all the inputs must be processed, so the time of execution will be the same as the full attack.

In Figure 11 are represented the two possible selection methods, and it is evident why selecting the outputs is better for a subset attack.

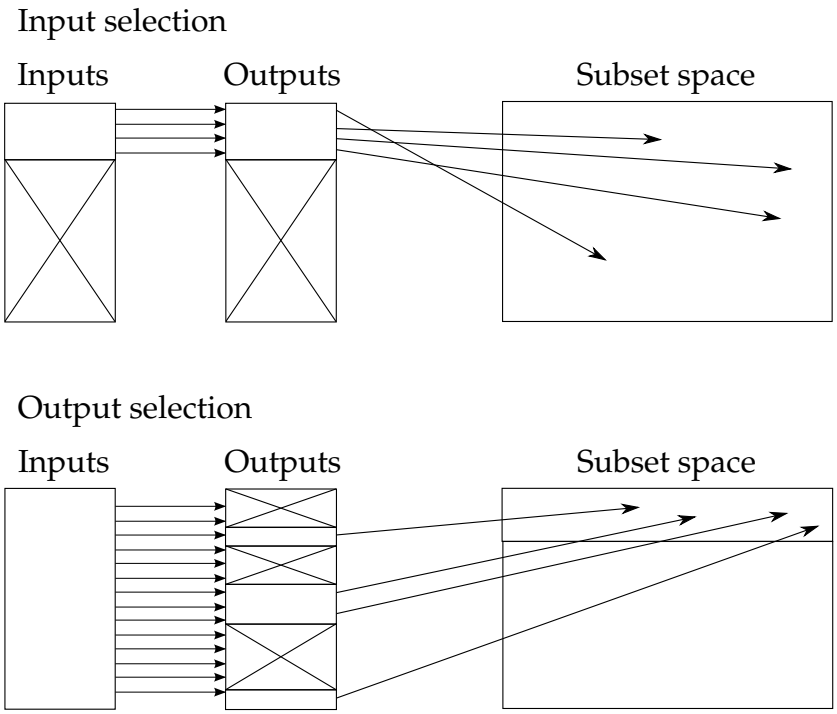

Figure 11: The inputs and outputs reduction method. The latter reduces the output space.

4.3.3 Memory optimization

As described in the previous section, the practical attack requires two data structures: the preallocated cache array and the sparse hash table. A small-size array generates too many false collisions, because many outputs will produce the same hash value. On the other hand the memory cost of a big-size array is a problem by itself, and cannot justify the size reduction of the hash table. To minimize the memory usage it is therefore necessary to balance these two data structures, but while the array size is known a priori, the hash table size must be estimated. To do so, three steps are necessary:

- Find the probability P that an array's bit will be written more than one time,
- Calculate the number of entries n the hash table has, based on p,
- Calculate the total size of the hash table based on n.

The first step requires the use of a Poisson distribution. A value x must be passed to the function, along with the average rate of success r, and the function will calculate the probability p that a generic value X is greater than or equal to x. Referring to our problem, p is the probability that a generic array cell X will be written at least x times. With $x = 2$, p is the probability that a generic array cell is written at least two times, thus adding a new entry to the hash table.

The average rate of success r is simply the number of inputs divided by the size of the array (in bits):

$$r = \frac{n_i}{|\text{array}|} \tag{24}$$

The second step requires to calculate the number of entries n to the hash table. This is the probability p multiplied by the number of array cells:

$$n = p \times |\text{array}| \tag{25}$$

Actually n is higher than that: Poisson calculate the probability that the number of outputs hashed to the same array cell is at least two, but if the number is, for example, three, two entries should be added instead of one. Since this doesn't influence the result so much, and these estimates don't need to be incredibly precise, this will be ignored, but the final memory usage could be a little more than expected. Finally, the total size of the hash table is n multiplied by the size of a single hash table entry:

$$|\text{hash table}| = n \times |\text{entry}| \tag{26}$$

With these informations it is possible to calculate the total memory usage in bytes:

$$|\text{array}| + P(r = \frac{n_i}{|\text{array}|}, x = 2, X \geq x) \times |\text{array}| \times |\text{entry}| \qquad (27)$$

The only variable parameter in Formula 27 is the size of the array. The other parameters are known, and for the attack to Keccak-512 previously described, the formula is:

$$|\text{array}| + P(r = \frac{2^{34}}{|\text{array}|}, x = 2, X \geq x) \times |\text{array}| \times 69 \qquad (28)$$

The array size that minimizes this value is 2^{38}, which occupies 2^{35} bytes, 32 GB. The total memory used amounts to 65 GB, which is a lot better than the 1104 GB required by the naive attack (which is actually a special case of the practical attack with $|\text{array}| = 2^0$). Figure 12 represents the memory required by the attack by varying the array size.

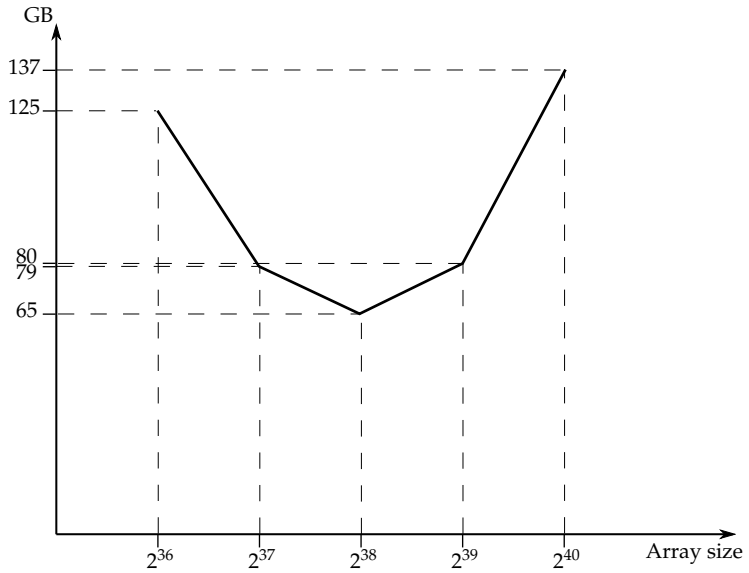

Figure 12: Memory usage of the attack with different array sizes.

On an average desktop PC the attack is still not practical, so we were forced to use the output reduction method described in the previous section. The number of inputs n_i in Formula 27 becomes a variable like the array size, and for each n_i it is possible to calculate the lower memory usage by varying the array size. The goal is to find the higher value of n_i that can lead to a memory usage low enough to fit inside the RAM. The

PC used for the attack had 16 GB of RAM, so n_i had to be reduced to one eight of the original value (still enough to obtain collisions, because with

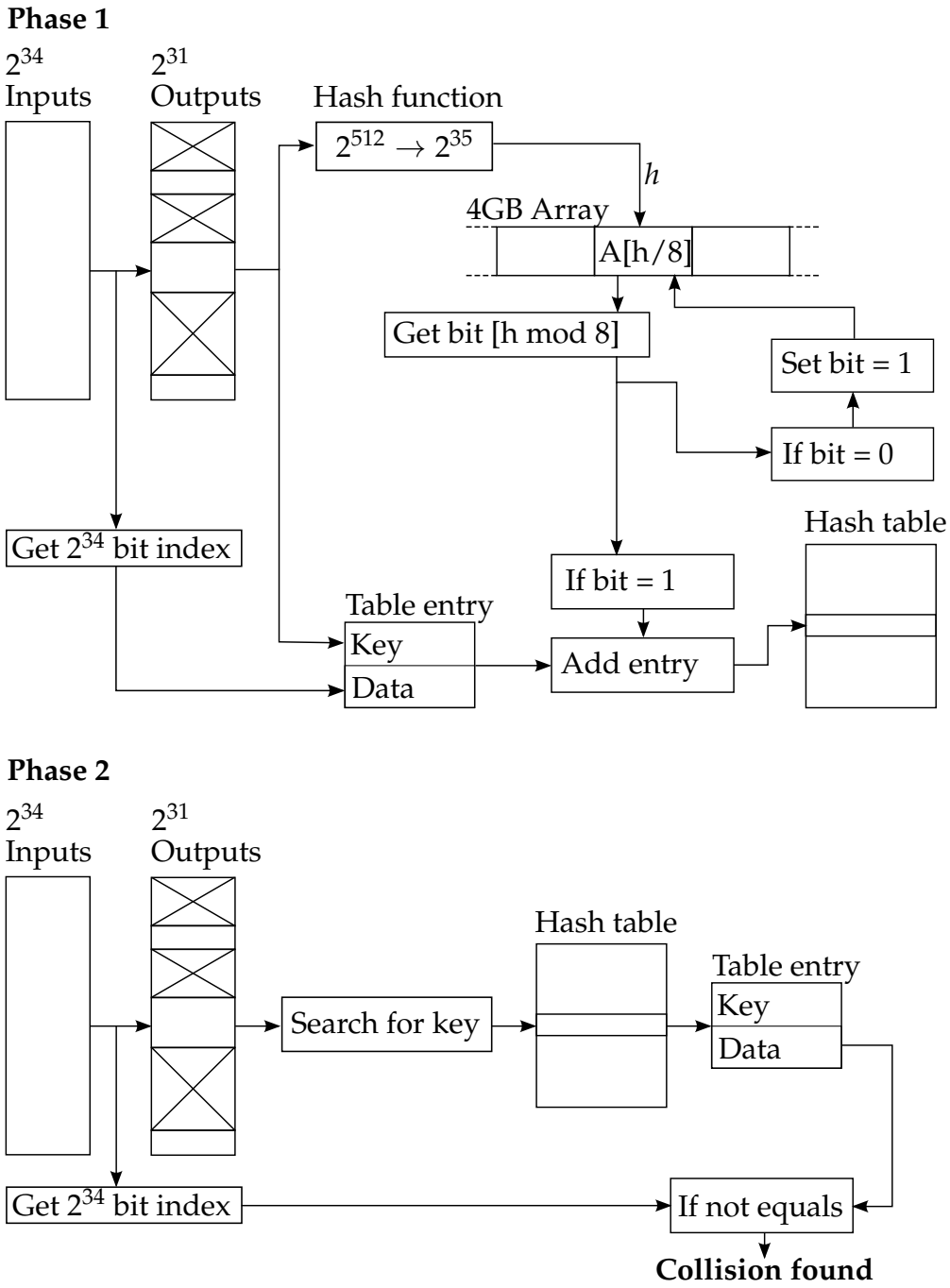

Figure 13: Structure of the practical three-rounds Keccak-512 attack.

the outputs reduction method the density of the output subset is not reduced).

The total memory usage after the outputs reduction is:

$$|\text{array}| + P(r = \frac{2^{34-3}}{|\text{array}|}, x = 2, X \geq x) \times |\text{array}| \times 69 \tag{29}$$

The value of $|\text{array}|$ that leads to the minimum memory usage is 2^{35}, 4 GB, with a practical total memory usage of 8.2 GB.

Figure 13 represents the whole structure of the attack, divided in the two phases, and the values reported are relative to the version of the attack divided in eight parts.

The attack was implemented in `Attack.c`, and the most relevant code is reported in Appendix 1.

4.3.4 Time optimization

The memory optimizations described in the previous section increase the time complexity of the algorithm:

· Two-steps attack: doubles the time

· Output reductions in n parts: multiplies the total time by n

To reduce the execution time of the attack as much as possible we used OpenMP directives to parallelize the code. The main loop of both steps of the attack has been divided between multiple threads, effectively reducing the time by k times, where k is the machine-dependent number of parallel threads. The use of the OpenMP directives is shown in the code excerpt of the attack in Appendix 1.

4.3.5 Results

With the optimizations described in the previous sections the attack was completed on a desktop PC in less than 8 hours, finding 15 collisions, the first one of them in less then 6 hours. Section 5.2 reports an example of collision. The attack was also run entirely on a remote server with 64 GB of RAM, reducing the outputs to one forth instead of one eight, thanks to the greater amount of RAM, and then running the four parts separately. The whole attack found 158 collisions, and each part took less than 5 hours to complete.

4.3.6 Other attacks

The authors of Keccak announced a "Crunchy Crypto contest" on the official Keccak website ([**11**]). It consists in a series of collision and preimage challenges on non-standard Keccak versions. These variants are generally easier to attack than the ones submitted to the SHA-3 competition, and the hash length is just 160 bits for each one of them, while the bitrate is $b - 160$. In this case b is in $\{200, 400, 800, 1600\}$, while in the standard versions it is always 1600. Also there is no padding in these variants.

We adapted the subset attack described in the previous section to two of these Keccak-160 variants, whose main difference is the Keccak-$f[b]$ algorithm they use:

· The first one uses Keccak-$f[400]$, $r = 240$, $c = 160$,

· the second one Keccak-$f[800]$, $r = 640$, $c = 160$.

From now on, to avoid confusion, I'll refer to these variants with the name of the Keccak-f algorithm that characterizes them: Keccak-$f[400]$ for the first one, Keccak-$f[800]$ for the second one, however they are both Keccak-160.

The first thing to do is find the i value that minimizes the number of inputs without reducing the probability of a collision. To do so it is necessary to find how many output lanes are occupied by the hash:

· Keccak-$f[400]$: $\frac{160}{400/25} = 10$,

· Keccak-$f[800]$: $\frac{160}{800/25} = 5$,

and how many lanes are occupied by the input:

· Keccak-$f[400]$: $\frac{240}{400/25} = 15$,

· Keccak-$f[800]$: $\frac{640}{800/25} = 20$.

Then, similarly to Formula 23, it is possible to calculate the i values:

· Keccak-$f[400]$: $2^{15*i} > \sqrt{2^{22+(10\times i)}}$,

· Keccak-$f[800]$: $2^{20*i} > \sqrt{2^{22+(5\times i)}}$.

So the optimal i values, the corresponding number of inputs n_i and the subspace size are:

· Keccak-$f[400]$: $i = 2$, with $n_i = 2^{30} > \sqrt{2^{42} = |\text{subset}|}$,

· Keccak-f[800]: $i = 1$, with $n_i = 2^{20} > \sqrt{2^{27} = |\text{subset}|}$.

It is interesting to notice that an high bitrate may lead to a worse security against this kind of attacks. The second variant uses Keccak-f[800] as its core algorithm, which should be more secure than using Keccak-f[400]. However the bitrate r is so high that with $i = 1$ there are already enough inputs to find a collision, and the output subspace is smaller too, because with an higher lane length w, the output hash is divided in bigger chunks and occupies less lanes.

The full attack to Keccak-f[400] required 30 minutes on a desktop PC and found more than two millions collisions, while the attack to Keccak-f[800] required less than one second, due to the very low number of inputs, and found ~ 2.000 collisions. These high numbers of collisions were to be expected, because in both cases the number of inputs is a lot greater than the square root of the output subspace.

Both these attacks didn't use the output reduction method, because the memory usage was not as high as in the Keccak-512 attack. Keccak-f[400] didn't even require the two-steps attack, because the number of inputs is so small, that every output can be saved directly to memory.

The code for both these attacks can be found in `Attack.c`. In Section 5.2 will be presented an example of collision for both these attacks.

5 Results

Section 5.1 reports the results obtained in trails generation. Hyphens will be used instead of zeroes to identify non-active bits, in order to better highlight the active ones.
Section 5.2 reports examples of collisions and the parameters used to obtain them.

5.1 Trails examples

Lowest-weight differential trail for three-rounds Keccak-f[1600] obtained with the naive method

Round 1:

---------- ---------- ---------- ---------- ----------
---------- ---------- ---------- 8--------- ----------
---------- ---------- ---------- ---------- ----------
---------- ---------- ---------- ---------- ----------
---------- ---------- ---------- 8--------- ----------

Round 2:

---------- ---------- ---------- ---------- ----------
---------- ---------- ---------- ---------- ----------
---------- ---------- ---------- ---------- ----------
---------- ---------- ---------- ---------- --8-------
---------- --4------- ---------- ---------- ----------

Round 3:

---------- ---------- -------2-------- ----------2--- ----------
----------1----- ---------- ---------- ---------- ---8-------
---------- 1---------- ----------2---- 8--------- ----------
---------- ---------- ---------- ----------2- ---1-------
--1------- ----8------ ---------- ---------- -1---------

Partial weights: 4,4,24.
Total weight: 32.

Lowest-weight differential trail for three-rounds Keccak-f[1600] obtained with the vortices method

Round 1:

----------2---- ---------- ---------- --2------- ----------
---------- ---------- ------4--------- --2------- ----------
---------- ---------- ---------- ---------- ----------
---------- ---------- ---------- ---------- ----------
----------2---- ---------- ------4--------- ---------- ----------

Round 2:

----------2---- ---------- ---------- ---------- ----------
----------2---- ---------- ---------- ---------- -------8--------
---------- ----1---------- ---------- ---------- -------8--------
---------- ---------- ---------- ---------- ----------
---------- ----1---------- ---------- ---------- ----------

Round 3:

----------2---- ---------- ---------- ---------- ----------
---------- --8------- ---------- ---------- ----------
---------- ---------- ---------- ---------- ----------
---------- --2------- --4------- ---------- ----------
---------- ---------- ----------4-- ---------- ----4----------

Partial weights: 12,12,12.
Total weight: 36.

Lowest-weight characteristic for 1.5 rounds of Keccak-f[1600]: zero-characteristic

Initial Difference:

```
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
```

After λ_0:

```
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
```

After χ_0:

```
---------------1 --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
--------------- --------------- --------------- --------------- ---------------
```

After λ_1:

```
---------------1 ----1---------- --------------- --------------- -----------8---
--------------- ---------2----- --------------- ----2---------- ---------------
---------------2 --------------- --------------- ------------2-- ---------------
--------1------- --------------- ------------4-- --------------- ---------------
-----1---------- --------------- --------------- --------------- --------------4
```

Total weight: 0.
Total weight after χ_1: 22.

5.2 Collisions examples

Three-rounds Keccak-512 attack

Parameters of Keccak-512:

· $n_r = 3$;

· $w = 64$;

· $b = 1600$;

· $l = 6$;

· $r = 576$;

· $c = 1024$;

· $|\text{hash}| = 512$;

Parameters of the attack:

· $i = 4$;

· $n_i = 2^{34}$;

· $n_o = 2^{31}$;

· outputs reduction = $1/8$;

· $|\text{array}| = 2^{35}$ b = 4 GB;

· $|\text{entry}| = 512 + 64 = 576$ b;

Results of the attack:

· Number of collisions found: 158.

· Execution time of one eight of the attack: 6 hours.

· Execution of the full attack: 2 days.

Example of a collision:

`Message 1:`

```
9999999999999999 EEEEEEEEEEEEEEEE 5555555555555555 5555555555555555 4444444444444444
BBBBBBBBBBBBBBBB DDDDDDDDDDDDDDDD EEEEEEEEEEEEEEEE DDDDDDDDDDDDDDDD 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

`Message 2:`

```
9999999999999999 EEEEEEEEEEEEEEEE 5555555555555555 9999999999999999 7777777777777777
AAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAA 8888888888888888 DDDDDDDDDDDDDDDD 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000
```

`Hash`

```
8111711111D1911F EF2767FF3C1FFFFC 7D67197536155454 A88A90B849088808 08C8EC0062A08006
77716507A513774F 99199B81A9FD99B1 577457777633754E ---------------- ----------------
```

Three-rounds Keccak-160 attack with Keccak-f[400] as core algorithm

Parameters of Keccak-160:

· $n_r = 3$;

· $w = 16$;

· $b = 400$;

· $l = 4$;

· $r = 240$;

· $c = 160$;

· $|\text{hash}| = 160$;

Parameters of the attack:

· $i = 2$;

· $n_i = 2^{30}$;

· $n_o = 2^{30}$;

· outputs reduction = 1;

· $|\text{array}| = 2^{35}$ b = 4 GB;

· $|\text{entry}| = 160 + 64 = 224$ b;

Results of the attack:

· Number of collisions found: 2092096.

· Execution of the full attack: 25 minutes.

Example of a collision:

Message 1:

```
0000 5555 AAAA 0000 0000
5555 0000 0000 0000 0000
0000 0000 0000 0000 0000
0000 0000 0000 0000 0000
0000 0000 0000 0000 0000
```

Message 2:

```
FFFF 5555 FFFF 5555 FFFF
0000 FFFF 0000 0000 FFFF
AAAA FFFF 0000 0000 AAAA
0000 0000 0000 0000 0000
0000 0000 0000 0000 0000
```

Hash

```
7F94 D9EB 3CEC 65FD 66B0
5C0C 69A7 FF4A CC4F DA32
```

Three-rounds Keccak-160 attack with Keccak-f[800] as core algorithm

Parameters of Keccak-160:

· $n_r = 3$;

· $w = 32$;

· $b = 800$;

· $l = 5$;

· $r = 640$;

· $c = 160$;

· $|\text{hash}| = 160$;

Parameters of the attack:

· $i = 1$;

· $n_i = 2^{20}$;

· $n_o = 2^{20}$;

· outputs reduction = 1;

· $|\text{array}| = 0$, not used.

· $|\text{entry}| = 160 + 64 = 224$ b;

Results of the attack:

· Number of collisions found: 2023.

· Execution of the full attack: < 1 second.

Example of a collision:

`Message 1:`

```
55555555 55555555 FFFFFFFF 55555555 FFFFFFFF
55555555 FFFFFFFF FFFFFFFF 55555555 FFFFFFFF
FFFFFFFF 55555555 FFFFFFFF FFFFFFFF 55555555
55555555 55555555 55555555 55555555 FFFFFFFF
00000000 00000000 00000000 00000000 00000000
```

`Message 2:`

```
55555555 55555555 FFFFFFFF 55555555 FFFFFFFF
FFFFFFFF 55555555 FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF 55555555 55555555 55555555 55555555
55555555 55555555 55555555 55555555 55555555
00000000 00000000 00000000 00000000 00000000
```

`Hash`

```
E527A3F0 0F2C8107 A93A9AAB F1E56CA3 72898C05
```

6 Conclusions

In the last few years a lot of papers regarding attacks on Keccak have come out. Most of them exploit weaknesses in possible implementations of the algorithm in order to reach a greater number of rounds. This work on the other hand is mainly focused on Keccak-f, and its goal is to test the security of the algorithm from differential attacks, both with trails generation and collision attacks to versions of Keccak with a simple sponge construction.

The subset attacks are very versatile and can break almost every three-rounds variant of Keccak, but they extend very poorly to the four-rounds scenario. The best and probably only chance to make them practical on four-rounds attacks is to generate a longer low-weight internal characteristic that spawns for two round and a half, but this task is made incredibly difficult due to the ι function, which breaks the internal symmetry of the state.

Simpler versions of Keccak, like Keccak-224 and Keccak-256, have been subject to attacks that practically reached four and almost five rounds, but Keccak-384 and Keccak-512 are still secure on that number of rounds. Considering that in the Keccak versions approved by NIST the number of rounds vary from 12 to 24, the main conclusion that can be drawn is that the standard Keccak variants are still incredibly secure at the time of this writing, and it is unlikely that practical attacks for the whole algorithms will ever be found.

However in the last few years a lot of different kinds of attacks (pre-image, key recovery, etc.) have been successfully implemented for non-standard Keccak sponge constructions (MAC, stream cipher, etc.). This is probably the direction to be taken for future studies on Keccak: since at the time of this writing Keccak-f is very secure and resistant to several types of attacks, the sponge constructions of each variant of Keccak must be studied to find eventual flaws and exploit them.

Appendix 1

This appendix contains the C implementation of the two-steps attack to the three-rounds version of Keccak-512. The following code often refers to functions or structures not defined in this appendix. Their purpose should be clear from the name and context, and the code is well commented, however their full implementation can be found in the full source code at [**14**].

First step

The first step of the attack stores possible collisions in the hash table.

```
//Pass 0-7 as argument to choose which
//part of the attack to execute
void keccak_512_three_rounds_attack(uint8_t fraction, FILE* file){

//Define or initialize variables
uint64_t i;           //Input index
uint64_t** cube;      //Input matrix
uint64_t** cube2;     //Auxiliary input matrix
uint64_t* hash_key;   //Hash of the output

//Initialize hash table
struct HashTable* ht = AllocateHashTable(64, 0);

//Preallocate cache array
uint64_t* buckets = calloc((uint64_t)536870912,sizeof(uint64_t));

if(buckets==NULL)
{
   fprintf(file,"Fatal: Out of memory.\n");
   return;
}

//Start loop with omp directives to parallelize the code
#pragma omp parallel private (cube,cube2,hash_key)
{
//Preallocate memory for each execution thread
cube = alloc_cube();
cube2 = alloc_cube();
hash_key = malloc(64);

//Main loop
#pragma omp for
for(i=0;i<17179869184;i++){
```

```
   //Insert a new input
   insert_cube(i,cube);

   //Calculate its output after three rounds
   three_rounds(cube,cube2,64);

   //Output reduction technique: keep only
   //outputs that end with the three bits
   //defined by the function argument
   if ((cube[0][0]&0x7) != fraction)  continue;

   //Extract hash from the output
   get_hash_from_cube(hash_key,cube);

   //Hash the hash with the second hash function.
   //This gives an index shorter than the whole hash
   uint64_t xxhash = XXH64 (hash_key, 64,0);

   //From the index we can get the position
   //inside the cache array, and the position
   //inside the single 64-bit word.
   uint64_t bucket_index = (xxhash&0x00000007ffffffc0)>>6;

   uint64_t bit_index = (uint64_t)1<<(uint8_t)(xxhash&0x3F);

   #pragma omp critical
   {
   //If the bit is free, set the bit
   if ((buckets[bucket_index]&bit_index)==0)
   {
        buckets[bucket_index]|=bit_index;
   }
   //If the bit is not 0, insert the input/output key
   //in the hash table. Instead of the whole input is stored
   //the index used to generate it.
   else
   {
      uint64_t* copied_hash = duplicate_hash(hash_key,7);
      HashInsert(ht, PTR_KEY(ht, copied_hash), i);
   }
   }
}
//Free thread memory
free_cube(cube);
free_cube(cube2);
free(hash_key);
}
```

Second step

The second step of the attack identifies the actual collisions among the possible ones.

```
//Free the cache array, no longer useful
fprintf(file,"\nFreeing buckets...\n");
free (buckets);

//Start step two
fprintf(file,"Searching for collisions...\n");
HTItem* item = NULL;

//Second parallel loop
#pragma omp parallel private (cube,cube2,item,hash_key)
{
//Preallocate memory used by each thread
cube = alloc_cube();
cube2 = alloc_cube();
hash_key = malloc(64);

#pragma omp for
for(i=0;i<17179869184;i++){

   //Insert new input
   insert_cube(i,cube);

   //Calculate its output
   three_rounds(cube,cube2,64);

   //Output selection technique: check if the outputs
   //ends with the chosen three bits
   if ((cube[0][0]&0x7) != fraction) continue;

   //Get hash from the output matrix
   get_hash_from_cube(hash_key,cube);

   //Query hash table for the output
   item = HashFind(ht, PTR_KEY(ht, hash_key));

   //If an item is found, and the inputs are different
   //print collision
   if(item!=NULL && item->data!=i)
   {
      fprintf(file,"[%ld,%ld]\n",item->data,i);
   }
   }
```

```
//Free thread memory
free_cube(cube);
free_cube(cube2);
free(hash_key);
}

//Clear and free hash table
ClearHashTable(ht);
FreeHashTable(ht);
}
}
```

References

[1] G. Bertoni et al. *The* KECCAK *reference*. http://keccak.noekeon.org/Keccak-reference-3.0.pdf(2011).

[2] Itai Dinur, Orr Dunkelman and Adi Shamir. *Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials*. Cryptology ePrint Archive, Report 2012/672. http://eprint.iacr.org/2012/672.pdf(2012).

[3] Joan Daemen and Gilles Van Assche. "Differential Propagation Analysis of Keccak". In: *Procs of FSE 12 LNCS 7549*. Springer, 2012, pp. 422–441. DOI: 10.1007/978-3-642-34047-5_24.

[4] G. Bertoni et al. KECCAK *implementation overview*. http://keccak.noekeon.org/Keccak-implementation-3.2.pdf(2012).

[5] Pawel Morawiecki et al. *Preimage attacks on the round-reduced Keccak with the aid of differential cryptanalysis*. Cryptology ePrint Archive, Report 2013/561. https://eprint.iacr.org/2013/561.pdf(2013).

[6] Donghoon Chang et al. *1st and 2nd Preimage Attacks on 7, 8 and 9 Rounds of Keccak-224,256,384,512*. http://csrc.nist.gov/groups/ST/hash/sha-3/Aug2014/documents/chang_paper_sha3_2014_workshop.pdf(2014).

[7] Itai Dinur, Orr Dunkelman and Adi Shamir. *New attacks on Keccak-224 and Keccak-256*. Cryptology ePrint Archive, Report 2011/624. https://eprint.iacr.org/2011/624.pdf(2011).

[8] Itai Dinur et al. *Practical Complexity Cube Attacks on Round-Reduced Keccak Sponge Function*. Cryptology ePrint Archive, Report 2014/259. http://eprint.iacr.org/2014/259.pdf(2014).

[9] Pawel Morawiecki and Marian Srebrny. *A SAT-based preimage analysis of reduced KECCAK hash functions*. Cryptology ePrint Archive, Report 2010/285. https://eprint.iacr.org/2010/285.pdf(2010).

[10] http://keccak.noekeon.org/index.html visited in october 2015.

[11] http://keccak.noekeon.org/crunchy_contest.html visited in october 2015.

[12] Itai Dinur et al. *Cube Attacks and Cube-attack-like Cryptanalysis on the Round-reduced Keccak Sponge Function*. Cryptology ePrint Archive, Report 2014/736. https://eprint.iacr.org/2014/736.pdf(2014).

[13] http://csrc.nist.gov/groups/ST/hash/sha-3/index.html visited in october 2015.

[14] http://www.club.di.unimi.it/people/ceffa.html visited in october 2015.