Practical Analysis of Reduced-Round Keccak ⋆

María Naya-Plasencia[1,3,⋆⋆], Andrea Röck[2,⋆⋆⋆], and Willi Meier[1,†]

[1] FHNW, Windisch, Switzerland
[2] Aalto University School of Science, Finland
[3] University of Versailles, France

Abstract. Keccak is a finalist of the SHA-3 competition. In this paper we propose a practical distinguisher on 4 rounds of the hash function with the submission parameters. Recently, the designers of Keccak published several challenges on reduced versions of the hash function. With regard to this, we propose a preimage attack on 2 rounds, a collision attack on 2 rounds and a near collision on 3 rounds of $\lfloor \text{Keccak} \rfloor_{224}$ and $\lfloor \text{Keccak} \rfloor_{256}$. These are the first practical cryptanalysis results on reduced rounds of the hash function scenario. All of our results have been implemented.

Keywords. hash function, Keccak, practical cryptanalysis, SHA-3

1 Introduction

Cryptographic hash functions are one of the three main branches of symmetric cryptography. They are deterministic functions, $\mathcal{H}$, that given an input or message M of an arbitrary length, return a short pseudo-random value of fixed length ℓ that must verify certain properties. This value must be easy to compute and is typically called digest or hash value, h. Hash functions have many important applications like authentication, integrity check of executables, digital signatures, etc. A hash function can normally be defined by an iterative construction and a compression function.

The digest should verify certain properties so that the hash function can be considered secure. The classical security requirements of a hash function are:

1. Collision resistance: finding two message M_1 and M_2 so that $\mathcal{H}(M_1) = \mathcal{H}(M_2)$ must be "hard". The generic collision attack, that applies to all hash functions, requires $2^{\ell/2}$ calls to the compression function.

⋆ This work was partially supported by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.

⋆⋆ Supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center of the Swiss National Science Foundation under grant number 5005-67322 and Partially supported by the French Agence Nationale de la Recherche through the SAPHIR2 project under Contract ANR-08-VERS-014.

⋆⋆⋆ Supported by the Academy of Finland under project 122736.

† Supported by GEBERT RÜF STIFTUNG, project no. GRS-069/07

2. Second preimage resistance: Given a message M_1 and its hash value $h = \mathcal{H}(M_1)$, finding another message M_2 so that $\mathcal{H}(M_2) = h$ must be "hard". The generic second preimage attack requires 2^ℓ calls to the compression function.
3. Preimage resistance: Given a hash value h, finding a message M_1 so that $\mathcal{H}(M_1) = h$ must be "hard". The generic preimage attack requires 2^ℓ calls to the compression function.

Defining what "hard" means in the previous concepts is a difficult task. In a strict way, we can ask that building a collision or a (second) preimage on the hash function must require at least as many calls to the compression function as the generic attacks. Beside these properties, a hash function must verify some other conditions, like for example generating hash values that are random-looking.

Recently, a big number of cryptanalysis results on hash functions have appeared, including the ones on the standards MD5 [13] and SHA-1 [12]. The confidence in the standard SHA-2 has then been undermined due to its resemblance with SHA-1. Because of this, the American National Institute of Standards and Technology (NIST) decided to launch in 2008 a competition to find a new hash function standard, SHA-3. From the 64 initial submissions, two rounds and three years later, only five candidates remain in the final round of this competition. One of them is Keccak.

Keccak is a sponge based hash function. The main cryptanalytic results published so far on Keccak are results on building blocks, that is, on the permutation involved in Keccak and not on the hash function. In [6], a zero-sum distinguisher on all 24 rounds of the permutation is proposed. This distinguisher has a complexity of 2^{1590}.

On the hash function setting, which is arguably a more interesting one, the only known results are marginally better than generic preimage attack for 6, 7 and 8 rounds [1] for the 512 bit version, having complexities of 2^{506}, 2^{507} and $2^{511.5}$ respectively. In [8], a practical preimage attack is proposed on three rounds of a modified Keccak, using different parameters than the recommended ones, like for example, a hash size of 1024 bits, which weakens considerably the hash function.

We believe that due to the lack of results on the hash setting of reduced rounds of the hash function, the authors of Keccak proposed a number of challenges for finding practical collisions and (second) preimages on reduced round versions of Keccak. Inspired by these challenges, we decided to study in detail the hash function setting, trying to find practical results. We present in this paper a distinguisher on the recommended hash functions $\lfloor \text{Keccak}[1088,512] \rfloor_{256}$ and $\lfloor \text{Keccak}[1152,448] \rfloor_{224}$ when reduced to 4 rounds, a second preimage on two rounds, a collision on 2 rounds and a near collision on 3 rounds. These are the first practical results of cryptanalysis of the Keccak hash function setting where all the parameters but the number of rounds remain unchanged. Note that the challenges proposed by the Keccak designers have smaller hash values (80 bits for a preimage, 160 bit for a collision) and a smaller capacity, $c = 160$, and are thus easier instances of the preimage and collision problem.

In [11], practical attacks on the compression functions of other hash functions were presented, but Keccak was not one of them. Our analysis methods are based on different techniques, and propose a deep study of reduced-round Keccak and its resistance to attacks on the hash function scenario, which are stronger results than compression function ones. For the sake of simplicity, in

Table 1. Best known cryptanalysis results on the Keccak hash function. We omit the analysis on the building blocks and detail all the results on the hash function setting.

Rounds	Version	Time	Memory	Generic	Type	Reference
6/7/8	512	$2^{506}/2^{507}/2^{511.5}$	–	2^{512}	Second Preimage Attack	[1]
4	256/224	2^{25}	–	2^{36}	Hash Function Distinguisher	Section 3
3	256/224	2^{25}	–	2^{64}	Hash Function Near-Collision	Section 4
2	256/224	2^{33}	–	2^{128}	Hash Function Collision	Section 5
2	256/224	2^{33}	2^{29}	2^{256}	Hash Function (Second) Preimage	Section 6

this paper we present the results on the recommended hash function with 256 bits of output, to which we refer at as Keccak during the analysis. The equivalent results for the 224 bit version are similar to them.

The paper is organized as follows: In Section 2, a description of Keccak and the notations that we use through the paper are given. Section 3 describes a differential distinguisher on 4 rounds of the recommended hash function Keccak-256, that is extended in Section 4 to a near-collision attack on 3 rounds. Section 5 presents a hash function collision on 2 rounds and Section 6 describes how to build a hash function (second) preimage for two rounds.

2 Keccak Description and Notations

Keccak is a family of sponge hash functions [4]. A sponge hash function absorbs a message block of r bits into its internal state and subsequently applies an internal permutation to the state. This step is repeated until the all blocks of the message to hash have been treated. Next, in the squeezing phase, r bits are generated from the state before each new permutation application, until the number of the wanted output bits has been generated. In the following we will describe the recommended Keccak versions for the SHA-3 competition, which are the ones that we have considered in our analysis. All the versions use the same internal permutation: Keccak$-f[1600]$.

The full Keccak$-f[1600]$ state is composed of 1600 bits, organized in 64 slices of 5×5 bits. The position of a bit in a slice can be given either by its x and y value or by its *bit*-number. The two notations are given in Table 2. The z coordinate gives the number of the slice $0 \leq z \leq 63$. Most of the steps in the round function of Keccak are invariant to a translation in z direction. The only part non-invariant is the round constant addition ι.

Table 2. Bit notation in a slice.

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	bit 1	bit 2	bit 3	bit 4	bit 5
$y=1$	bit 6	bit 7	bit 8	bit 9	bit 10
$y=0$	bit 11	bit 12	bit 13	bit 14	bit 15
$y=4$	bit 16	bit 17	bit 18	bit 19	bit 20
$y=3$	bit 21	bit 22	bit 23	bit 24	bit 25

The bits in the state can be also numbered from 0 to 1599. The conversion from x, y, z coordinate to global state bit position is done as follows:

$$\text{global pos} = 64(5y + x) + z.$$

The inner permutation of the full Keccak hash function consists of 24 iterations of the round function. The round function itself is composed of five steps:

1. θ: Xor to each bit the XOR of two *columns* (column = same x value, y from 0 to 4). The first column is in the same slice as the bit and the second column is in the slice before the bit.
2. ρ: Translate a bit in z direction.
3. π: Permute the bits within a slice.
4. χ: Apply a 5×5 S-box on one row (row = same y value, x from 0 to 4).
5. ι: Addition of round constant.

Each of the versions (224, 256, 384 and 512 output bits) has a different block message size r. The capacity in a sponge construction is the size of the internal state minus the size of a message block. Consequently, they all have a different capacity c:

- For an output of 224 bits, $r = 1152$ and $c = 448$.
- For an output of 256 bits, $r = 1088$ and $c = 512$.
- For an output of 384 bits, $r = 832$ and $c = 768$.
- For an output of 512 bits, $r = 576$ and $c = 1024$.

For more details we refer to [3]. As previously said, in this paper we analyze the 256 bit version, that we will denote simply by Keccak in the following. All of our results can be directly extended to the 224 bit-output version.

3 Differential Distinguisher

In this section we first present an efficient way of searching low weight differential paths. This method was used to find the differential paths that we apply for the distinguisher on 4 rounds, the collision on 2 rounds and the near collision on 3 rounds. For the distinguisher we use in addition the concept of free bits as was used in [7].

3.1 Searching Differential Paths

As the authors define in [2], a state-difference is a *kernel* if it is invariant to the function θ, e.g. in each column we have a difference in zero or in an even number of bits. If we have a column where we have a difference in an odd number of bits, θ will spread this difference to 10 bits. Thus, for a low weight differential path we would like the state-differences to stay a kernel as long as possible. The designers of KECCAK show in [2] that it is not possible to construct low weight differentials that are a kernel for three states in a row, however two states in a row is possible, though they are not given in the documentations. We will denote the two kernels in a row a *double kernel*.

For our search we use the special property of χ that every 1-bit difference in a row constructed before χ will produce the same 1-bit difference after χ with probability 2^{-2}. Thus such a 1-bit difference will be invariant to the only non-linear part with probability 2^{-2}. If in addition we have a kernel, *i.e.* the difference is invariant to θ, we can concentrate on the functions ρ and π to find a double kernel.

For finding a double kernel we use the following procedure. At first, we fix in how many slices we want a difference in the first state. An example for a kernel in 3 slices is given in Figure 1. We start by choosing one bit in slice $z = 0$. The following algorithm will be repeated for all bits in slice $z = 0$. From our chosen bit we compute the position after one application of ρ and π. For this new bit position we check all bits in the same column and compute its position back by applying π^{-1} and ρ^{-1}. We once again check all possible bits in the same column and compute their position after applying ρ and π. We continue this procedure until we have touched the wanted number of slices. We will find a double kernel if after the last step we are again at the original slice at the right column.

This basic method allows us to find all double kernels which have k active slices in each of the two kernels with a complexity of $25 * 4^{2k-1}$. Every solution will be found $2 * k$ times, since every point of the first kernel can be a starting point.

By this method we can find very fast all possible differential paths that are a kernel for two states in a row and have low hamming weight. We can use this method for example to find a differential path over 4 rounds which has a probability of 2^{-142} and a double-kernel on 3 slices in the first two non-linear layers, and a differential path over 5 rounds with a probability of 2^{-510} by computing one step back and two steps forwards from a double-kernel on 6 slices. However, the objective of our paper is not to find the best differential path. We used the method to find suitable differential paths for our analysis.

3.2 Conditional Differentials and Free Bits

In [7], the concept of conditional differentials applied to NLFSR-based systems is introduced. The main idea is to consider a differential path with a good probability and try to control the first rounds of the path by imposing some conditions

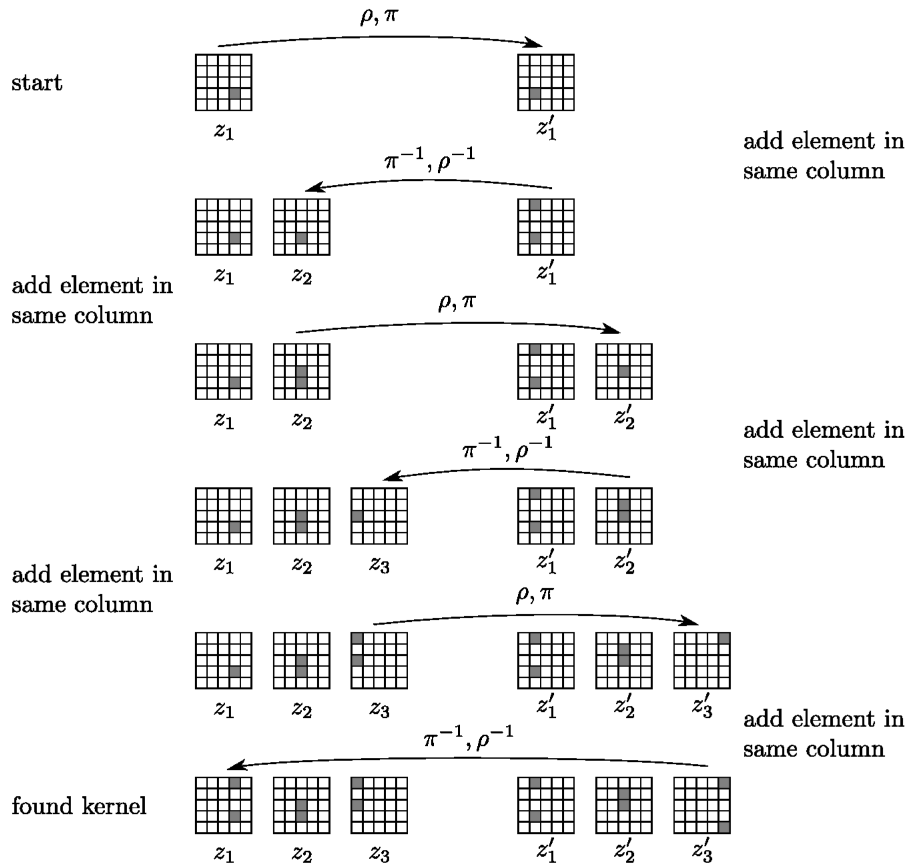

Fig. 1. Successful search of a double kernel on 3 slices.

on the values of the internal state, so that the path with these conditions is verified with probability 1. For detecting a bias after the biggest number of rounds possible, the concept of free bits is introduced. In [10,5], some related work can be found.

The free bits [7] are the input bits such that, once we have found a pair of inputs that satisfies the differential path for the first rounds and so the corresponding needed conditions, can take any values without contradicting the conditions needed to satisfy the differential path. This means that once we have found a pair of inputs that follows the differential characteristic, we can change these bits and the new pairs will still follow the characteristic for these these first rounds.

3.3 Best Differential Paths

We have tested all low weight differential paths starting with a double kernel and checked which of them have an input difference that fits into the message part of the hash function. This is needed in order to perform a distinguisher in the hash function, as the difference will be introduced by the last round message block. Each path has a probability of being verified, i.e. the probability a random pair following the characteristic, that is given by the χ transformations behaving as needed. The highest probability of 2^{-24} was achieved by two paths, both having characteristic of 6-6-6 active S-boxes. In both cases, the difference of the first two rounds are a kernel and the first difference fits into a 1088 bit message. We chose the one which had less active bits one round later. This path stays valid for any translation in the z direction.

The best path: Let us define the following differences:

$$\begin{aligned}
\Delta_1 : & x = 0,\, y = 0,\, bit = 13,\, z = 0 \\
& x = 0,\, y = 1,\, bit = 8,\ \ z = 0 \\
& x = 2,\, y = 1,\, bit = 10,\, z = 30 \\
& x = 2,\, y = 2,\, bit = 5,\ \ z = 30 \\
& x = 1,\, y = 0,\, bit = 14,\, z = 63 \\
& x = 1,\, y = 2,\, bit = 4,\ \ z = 63
\end{aligned}$$

$$\begin{aligned}
\Delta_2 : & x = 0,\, y = 0,\, bit = 13,\, z = 0 \\
& x = 0,\, y = 2,\, bit = 3,\ \ z = 0 \\
& x = 2,\, y = 0,\, bit = 15,\, z = 9 \\
& x = 2,\, y = 3,\, bit = 25,\, z = 9 \\
& x = 1,\, y = 2,\, bit = 4,\ \ z = 36 \\
& x = 1,\, y = 3,\, bit = 24,\, z = 36
\end{aligned}$$

$$\begin{aligned}
\Delta_3 : & x = 0,\, y = 0,\, bit = 13,\, z = 0 \\
& x = 2,\, y = 1,\, bit = 10,\, z = 3 \\
& x = 0,\, y = 4,\, bit = 18,\, z = 7 \\
& x = 3,\, y = 1,\, bit = 6,\ \ z = 17 \\
& x = 3,\, y = 3,\, bit = 21,\, z = 24 \\
& x = 2,\, y = 3,\, bit = 25,\, z = 46
\end{aligned}$$

Then the path is the following (where we ignore ι since it is not important for the differences):

$$\Delta_1 \overbrace{\xrightarrow{\theta,\rho,\pi,} \Delta_2 \xrightarrow{\chi} \Delta_2}^{\text{round}} \overbrace{\xrightarrow{\theta,\rho,\pi,} \Delta_3 \xrightarrow{\chi} \Delta_3}^{\text{round}}$$

3.4 Distinguisher on 4 Rounds of the Hash Function

Let $\mathcal{H}^{(4)}$ be the Keccak-256 hash function reduced to 4 rounds and f_{R} one round of the Keccak$-f[1600]$ function. For a partial message M, we define by X_M the internal state of Keccak after absorbing M.

In a first off-line step we find a padded message $M||m$ such that $(X_M \oplus m, X_M \oplus m \oplus \Delta_1)$ will follow the differential path described in the previous section, *i.e.*

$$f_{\mathrm{R}}^2(X_M \oplus m) \oplus f_{\mathrm{R}}^2(X_M \oplus m \oplus \Delta_1) = \Delta_3 \ .$$

Such a message can be found by trying 2^{24} random messages. We recall that m and $m \oplus \Delta_1$ are the last message blocks including the correct padding.

Next we check how many *free bits* we have for our differential path within the range of the r message bits. We recall that by a free bit we mean a bit that we can change in m, so that the differential path is still verified. We find the following results:

- For the differential path described in Section 3.3, there are **81 free bits** within the $r = 1088$ bits of the message block. They are listed in Appendix A.1.

We can now define the vectors pace $A \subset \mathbb{F}_2^r$ containing all binary vectors of size r which are zero at all non-free bit positions, i.e. A has dimension 81. This means, for any $(X_M \oplus m, X_M \oplus m \oplus \Delta_1)$ which follows the differential path and any difference $\alpha \in A$, the pair of states $(X_M \oplus m \oplus \alpha, X_M \oplus m \oplus \Delta_1 \oplus \alpha)$ will also follow the differential path.

For our distinguisher we have to define what we mean by a bias. Let $\{0^r\}$ be the set containing the all-zero vector of size r bits.

Definition 1. *Let $\{\alpha_1, \ldots, \alpha_N\}$ be a set of N distinct differences, $\alpha_i \in A \backslash \{0^r\}$, and Δ a single difference. We denote by M the initial message with a length of some message blocks, and m be the final message part, such that m including the final padding fits into one message block. The concatenation of M and m is denoted as $M\|m$. Let $\mathcal{H}(M\|m)_i$ define the ith bit of the hash of $M\|m$ computed using the function $\mathcal{H}$. Then we define the bias of the ith bit as*

$$\epsilon_i^{\mathcal{H}} = \frac{\#\{1 \leq j \leq N : \mathcal{H}\Big(M\|(m \oplus \alpha_j)\Big)_i \oplus \mathcal{H}\Big(M\|(m \oplus \alpha_j \oplus \Delta)\Big)_i = 1\}}{N} - \frac{1}{2}$$

We will use the following property for our distinguisher:

Distinguishing property: For any M such that $(X_M \oplus m, X_M \oplus m \oplus \Delta_1)$ follows the differential path described in Section 3.3, any set of distinct differences $\{\alpha_1, \ldots, \alpha_N\}$, $\alpha_j \in A \setminus \{0^r\}$, $\Delta = \Delta_1$ and $\mathcal{H} = \mathcal{H}^{(4)}$ being the 4 round version of the Keccak hash function, there are **18 positions i in the hash** where the absolute value of the bias is $|\epsilon_i^{\mathcal{H}^{(4)}}| = 2^{-1}$ for $N \geq 1$. The exact positions and their bias is listed in Appendix A.2

3.5 Implementation of the Distinguisher

In a first off-line step we search a message $M||m$ such that $(X_M \oplus m, X_M \oplus m \oplus \Delta_1)$ follows the differential path. We have to try 2^{24} messages and for each of them compute first X_M by absorbing the message blocks in M and then two rounds of the round-function on $X_M \oplus m$ and on $X_M \oplus m \oplus \Delta_1$, to check if the differential path has been verified. Thus this step costs 2^{25} in time.

The next step is the online step, where we will determine if $\mathcal{H}$ is a random function or the 4 round version of Keccak. We choose randomly a set of distinct differences $\{\alpha_1, \ldots, \alpha_N\}$, $\alpha_i \in A \setminus \{0^r\}$, and test the bias at the 18 predefined positions for Δ_1. In the case of Keccak the bias will be correct for any $N \geq 1$, in the case of a random oracle this will happen with a probability of 2^{-18N}. Thus we can distinguish the 4-round version of Keccak with an off-line precomputation complexity of 2^{25} and an on-line complexity of $2N$ where $N \geq 1$.

Remark 1. The previous test has a complexity of $2^{25} + 2N$ and searches in a precomputation phase for a suitable message M. Without this precomputation phase, we can still have a distinguisher with a cost of $2^{25}N$ where we need $N \geq 2$. For this we define a test $\mathcal{T}$, in the following way:

– A message M passes the test $\mathcal{T}$ if for Δ_1 and a set of distinct differences $\{\alpha_1, \ldots, \alpha_N\}$, $\alpha_i \in A \setminus \{0^r\}$, its bias has the correct value in the 18 positions defined in the previous section.

In the random case, a message M passes the test with probability 2^{-18N}, in the case of a 4-round Keccak and a message where X_M follows the differential path, this happens with probability 1. We will find such an M with probability 2^{-24}, thus for the 4-round Keccak the total probability of finding an M passing the test is $2^{-24} + 2^{-18N}$. For $N \geq 2$ the first term is dominating and by testing 2^{24} messages we are able to distinguish the 4-round Keccak from a random oracle. The total time complexity is $2^{24} * 2N$.

4 Near-Collisions for 3 Rounds on the 256-bit Hash Function

In this section we show how to build near-collisions on the 3-round hash function by using the previous path. As we said before, for verifying the first two rounds and having 6 bit-differences after them, there are 24 conditions that need to be verified. This will happen with a probability of 2^{-24}. Thus, when inserting 2^{24} well padded message blocks from a fixed and chosen chaining value we can assume to find one that follows the differential path for two rounds.

We now study what happens one round later. For this we apply the linear part, θ, ρ and π, on the difference Δ_3. We call this new difference the state S.

The output of the hash function is given by the lanes of 64 bits at positions $y = 0$ and $x = 0, 1, 2, 3$. This means that if we have a difference in the hash value, we must have at least one difference with $y = 0$ in the corresponding

slice before the final χ transformation. We have checked in state S which slices have differences in the middle lines, so in $y = 0$. This happens for 13 slices, and each of them contains just one difference in $y = 0$. Amongst them, four slices contain a difference in $x = 4$, which does not belong to the hash output. If such a difference passes χ in such a way that the 1-bit output difference equals the 1-bit input difference, we won't have a difference in the digest. If not, the 1-bit input difference might generate up to two bits of differences in the hash. The remaining $13 - 4 = 9$ slices have differences that will produce, each, at least a 1-bit difference in the hash value, so it will always have at least 9 differences after two rounds. Besides that, 6 amongst these differences can generate one more bit of difference, and 3 can generate up to two more difference bits, depending on χ.

At this stage, we can do two things:

1. Not increasing the complexity, and finding near-collisions for the bits that we know for sure won't have a difference. This number of bits is $256 - 4*2 - 6*2 - 3*3 = 227$. In this case, we can build near collisions of 227 bits with a complexity of 2^{24}, while the generic complexity would be $\left(\frac{2^{256}}{\binom{256}{227}}\right)^{1/2} = 2^{64}$. This has been verified experimentally, and an example is given in Appendix B.
2. Try to control the additional conditions and just have the inevitable 9 differences in the hash value. As we saw before, this means that we have to control $29 - 9 = 20$ additional bit conditions. These are the conditions given by χ so that it does not spread the already existing differences. The total complexity for having a near-collision on $256 - 9 = 247$ bits is $2^{24+20} = 2^{44}$, while the generic complexity is $\left(\frac{2^{256}}{\binom{256}{247}}\right)^{1/2} = 2^{101}$.

5 Hash Function Collisions on 2 Rounds

To find a collision on the hash function by means of a differential path we need to find a path that fits into the message and has no difference in the hash. This is not possible by a double kernel on three slices, however we found a suitable path considering double kernels on four slices.

$$
\begin{aligned}
\Delta_1 : & x = 1, y = 2, bit = 4, \quad z = 0 \\
& x = 1, y = 3, bit = 24, z = 0 \\
& x = 0, y = 2, bit = 3, \quad z = 4 \\
& x = 0, y = 3, bit = 23, z = 4 \\
& x = 4, y = 0, bit = 12, z = 35 \\
& x = 4, y = 2, bit = 2, \quad z = 35 \\
& x = 1, y = 0, bit = 14, z = 61 \\
& x = 1, y = 2, bit = 4, \quad z = 61
\end{aligned}
$$

$$
\begin{aligned}
\Delta_2 : \; & x = 2,\, y = 1,\, bit = 10,\, z = 7 \\
& x = 2,\, y = 3,\, bit = 25,\, z = 7 \\
& x = 2,\, y = 3,\, bit = 25,\, z = 10 \\
& x = 2,\, y = 4,\, bit = 20,\, z = 10 \\
& x = 3,\, y = 1,\, bit = 6,\;\; z = 45 \\
& x = 3,\, y = 4,\, bit = 16,\, z = 45 \\
& x = 0,\, y = 2,\, bit = 3,\;\; z = 62 \\
& x = 0,\, y = 3,\, bit = 23,\, z = 62
\end{aligned}
$$

$$
\begin{aligned}
\Delta_3 : \; & x = 2,\, y = 1,\, bit = 10,\, z = 1 \\
& x = 4,\, y = 1,\, bit = 7,\;\; z = 7 \\
& x = 1,\, y = 2,\, bit = 4,\;\; z = 13 \\
& x = 3,\, y = 3,\, bit = 21,\, z = 22 \\
& x = 3,\, y = 3,\, bit = 21,\, z = 25 \\
& x = 1,\, y = 4,\, bit = 19,\, z = 36 \\
& x = 4,\, y = 3,\, bit = 22,\, z = 37 \\
& x = 3,\, y = 4,\, bit = 16,\, z = 39
\end{aligned}
$$

We will use the following path:

$$
\Delta_1 \overbrace{\xrightarrow{\theta,\rho,\pi,} \Delta_2 \xrightarrow{\chi} \Delta_2}^{\text{round}} \overbrace{\xrightarrow{\theta,\rho,\pi,} \Delta_3 \xrightarrow{\chi} \Delta_3}^{\text{round}}
$$

The differences Δ_2 and Δ_3 have each eight rows with a 1-bit difference in the input and in the output of χ. This lead to a total probability of 2^{-32} of following the differential characteristic. The difference in Δ_1 lies in the message block and difference in Δ_3 lies outside of the hash value. If we try random messages pairs where we introduce Δ_1 in the last message block, we have a probability of 2^{-32} that the last two rounds follows our differential path, which ensures that we have a collision in the final hash. Therefore, the complexity of this 2-round collision is 2^{33}. In practice, we find a collision much faster, after around 2^{13} steps. We give an example of such a collision in Appendix C.

6 Practical (Second) Preimages on 2 Rounds of the 256-bit Hash Function

In this section we present a (second) preimage attack for a reduced version with two rounds of Keccak . The preimage works with a complexity of about 2^{33} in time and 2^{29} in memory. It also applies to several of the challenges with other parameters, but we present here the detailed case of 2 rounds of the recommended version for SHA-3. An example for a preimage can be found in Appendix D.

6.1 Main Scheme

Figure 2 gives a representation of the (second) preimage attack on 2 rounds. In it, we can see the slices at different states. In each slice, the square represents a 64 bit lane. The white lanes are known and the colored ones are not. For the sake of simplicity, we will omit the ι transformation in the explanation, as it does not affect the procedure of the attack, but it must be taken into account when implementing the attack.

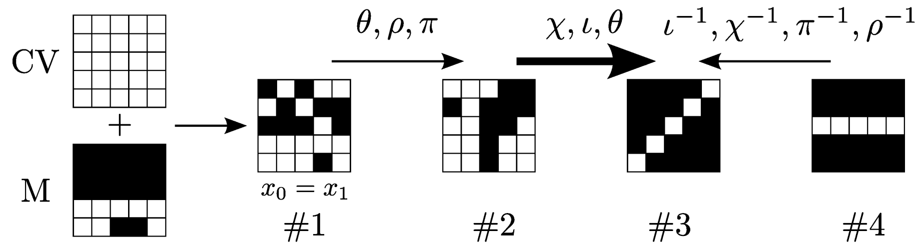

Fig. 2. Diagram of the 2-round preimage attack. Each square represents a 64 bit lane. Each white lane is a lane known and fixed, each colored one, a not-yet-fixed lane.

We are given a hash value, which is 4 out of the 5 white lanes in the most right slice #4, in Fig. 2, that represents the final state after the permutation. The fifth lane is not known but we can choose a random value for it and fix it. What we want to find now is, given a chaining value, for example the initial one, a message block that produces the values of these five lanes, and so the initial given hash value corresponding to 4 out of these 5 lanes.

In Fig. 2, the gray lanes show into which lanes of the chaining value the message is xored. The lanes marked with a zero are the lanes of the message that we are going to fix to zero. The lanes marked with $(a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1)$ are the parts of the message that we do not fix until the end of the attack. The only condition we ask from them is that: $a_0 = a_1; \ldots; e_0 = e_1$. In a generic way we will say $x_0 = x_1$. These conditions are asked for so that the first operation θ will not change the unknown lanes. From the initial state #1 we can then compute the known lanes in #2 after θ, ρ and π, as well as the positions of the unknown lanes. Figure 3 shows the movement of the bits in detail. Imposing the previous conditions we still have $5 * 64$ degrees of freedom for the message, which is the same as the number of bits in which we want to collide in the end. We can then expect to find one solution.

In the backward direction, we can invert from #4 the known white line of five lanes in the final state with χ^{-1}. Then, we can apply the inverse of π and of ρ and obtain the values and positions of the 5 known lanes in #3.

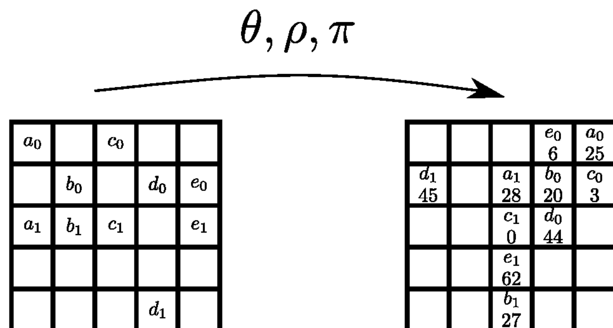

Fig. 3. Shows how the bits $a_0, \dots, e_1$ get moved by θ, ρ, π. The number k under x_i means that x_i at slice z on the left side gets moved to slice $z + k \pmod{64}$ after the transformation.

Then, the issue is to find the values of the ten 64-bit words $(a_0, a_1, b_0, b_1, c_0, c_1, d_0, d_1, e_0, e_1)$ in #2 that make possible the transition by the operations χ and θ, from the state #2, that we have obtained computing forward, to the state #3, that we have obtained computing backward. For this we are going to start by finding the bits that verify the relations for a few slices. The idea is, for example, to consider first groups of three slices where we guess all the involved bits of $a_0, \dots, e_1$, and next we can do a sieving by just keeping the guesses ones that produce by χ and θ the values of the $5 \times 2 = 10$ known bits from #3 in the middle and last slices of the group of three slices. This is possible as for computing the output of θ in a specific slice, we need to know this same slice and the previous one in the input state. We say that one bit (from x_0 for example) is *repeated* in a group of slices when the bit from x_1, corresponding to the same slice in the initial difference, also appears in this group of slices.

6.2 Finding Partial Solutions

Partial solutions for 3 slices: We start by finding partial solutions for groups of three slices as previously described. In three slices there are 3×10 bit-variables from $a_0, \dots, e_1$. If we consider the conditions $x_0 = x_1$ there are two variables that are repeated, so we have to guess in total $30 - 2 = 28$ variables. The two repeated values come from the values of ρ for d_0 and d_1, as there is a translation of 1 in between them. Out of the three bits of d_0 intervening in three consecutive slices, 2 of them will be equal to 2 of the bits of d_1 intervening in these same slices. For each guess we check if the 10 bits already fixed computing backwards on the two last slices collide, which will happen with a probability of 2^{-10}. In total, for each group of three slices that we try to find partial solutions for, we obtain $2^{28-10} = 2^{18}$ solutions. We will repeat this for 16 consecutive groups of 3 slices, leaving 16 out. By using the methods described in Section 6.4, the time

complexity of building the list is given by the size of the list. Thus this step has a time and memory complexity of $16 * 2^{18} = 2^{22}$.

Partial solutions for 6 slices: We are going to merge here each two consecutive groups of three slices. We have 2^{18*2} possibilities, but, because of the conditions $x_0 = x_1$, a group of three slices has 7 repeated variables regarding a consecutive group. Also, when we merge two groups, we can check if we obtain the wanted values on five more bits from the output (from the first slice of the second group). This way we obtain $2^{18*2-7-5} = 2^{24}$ solutions for each one of the 8 groups of 6 slices. The bottleneck of this step is the number of solutions, 8×2^{24} in both time and memory, as the merge of two lists can be done using the instant matching algorithm described in [9] by using the methods in 6.4. This algorithm can also be applied in the next steps, so in all the cases, the bottleneck will be the number of solutions obtained. This step has a time and memory complexity of $8 * 2^{24} = 2^{27}$.

Partial solutions for 12 slices: The same way, we merge here each two consecutive groups of 6 slices for generating 4 groups of 12 slices. In this case the number of repeated bits in the merge is 16, so the total number of solutions that we will obtain is $2^{24*2-16-5} = 2^{27}$. This step has a time and memory complexity of $4 * 2^{27} = 2^{29}$.

Partial solutions for 24 slices: We merge here each two consecutive groups of 12 slices for generating 2 groups of 24 slices. In this case the number of repeated bits in the merge is 22, so the total number of solutions that we will obtain is $2^{27*2-22-5} = 2^{27}$. We have a time and memory complexity of $2 * 2^{27} = 2^{28}$.

Partial solutions for 48 slices: Finally, we merge the 2 groups of 24 slices, for obtaining one group of solutions for 48 consecutive slices. In this case, the number of repeated bits due to conditions is also 22, so we obtain 2^{27} solutions. In these 48 slices we have determined 480 bit-variables from $a_0, \ldots, e_1$, and there are a total of

$$2 * 16 + 7 * 8 + 16 * 4 + 22 * 2 + 22 = 218$$

repeated variables amongst them. This means that there are $480 - 218 * 2 = 44$ bit-variables not repeated, that are then repeated in the 16 slices that we have yet to treat. This step has a time and memory complexity of 2^{27}.

Partial solutions for 16 slices: For finding solutions for 16 slices we first find solutions for the 12 rightmost slices the same way as before, and obtaining 2^{27} partial solutions. Let us remark here that in 12 slices there are 120 bit-variables fixed and 38 out of them are repeated ones. This means that there are 44 variables not repeated that must have their corresponding bit-variable in the 48-slice group or in the remaining 4-slice group.

Next, we can obtain solutions for the 4 remaining and consecutive slices, where we have 40 bit-variables, and 5 of them are repeated. Additionally, we have to collide on 5×3 bits that we can compute of the output. This leaves us

with $2^{40-5-15} = 2^{20}$ solutions. As there are 40 bit-variables and 5 are repeated, there are $40 - 5 * 2 = 30$ variables not repeated.

We know that in the 48-slice group, there are 44 bit-variables that must be on either the independent group of 12 or on the groups of 4 slices. This means that out of the $44 + 30 = 74$ not-repeated variables in these last two groups, 44 will correspond to the ones in the 48-slices group. Then, of the 30 remaining variables, half must be in the groups of 12, and 15 must be in the group of 4. This can also be seen as the following system of equations, where A represents the number of common bits between the 48-slice group and the 4-slice group, B is the number of common bits between the 48-slice group and the 12-slice group, and C is the number of common bits between the 4-slice group and the 12-slice group. From the previous numbers we know that $A + B = 44$, $B + C = 44$ and $A + C = 30$. Then we have $A = 15$, $B = 29$ and $C = 15$. The number of additional conditions determined by the repeated bits that we have for merging the group of 4 and the group of 12 slices is then $C = 15$.

We can then merge the group of 12 slices and the one of 4 obtaining

$$2^{27+20-15-5} = 2^{27} \text{ solutions.}$$

This step has a time complexity of $2^{27}+2^{20}+2^{27} \approx 2^{28}$ and a memory complexity of 2^{27}.

6.3 Matching 48 Slices with 16 Slices:

Now, we can match the just obtained group of 16 slices with the one of 48. As we said, they have 44 variables in common where they have to collide, and there are $2 * 5$ bits of the output that will also be determined. This leaves us with

$$2^{27}2^{27}2^{-44}2^{2*5} = 1 \text{ solution,}$$

as expected. This step has a time complexity defined by the size of one input list and is thus 2^{27}. During the attack we keep only the latest generated lists in the memory, thus the the memory complexity is bounded by 2^{29}. The time complexity is given by $2^{22} + 2^{27} + 2^{29} + 2^{28} + 2^{27} + 2^{28} + 2^{27} \approx 2^{31}$ Thus, we can obtain a (second) preimage with a complexity of 2^{31} in time and 2^{29} in memory.

6.4 Implementation Remarks

Two methods can help in an efficient implementation of the attack. Let us assume we want to merge the block from slice i to j with the block from slice $j+1$ to k. We first precompute a list containing all solutions for merging slice j and slice $j+1$. We have 10 bits in each of the two slices, 1 repeated bit and 5 conditions from the output, thus we have in total 2^{14} solutions that we sort by the 2^{10} values in slice j. The costs of building this list is negligible in comparison to the remaining time complexities. Next, for each solution in the first block (i to j) we compute the values of the bits that will repeat in the second block. We will

sort the solution in this first block by the value of the slice in j and the values of the repeated bits. We do the same thing for the second block ($j+1$ to k) and sort it by the value of slice $j+1$ and the values of the repeated bits. Now we can easily merge the two lists using the precomputed list of matches from slices j to $j+1$.

6.5 Dealing with the Padding

In Keccak, the message is not padded with the length, but with a simple padding where to the last message block we append a 1, a number of 0's and another 1 so that it completes the final block. For us to obtain a valid message block that fits into the last block, we need to have a 1 in the last position and a zero in the previous one. We have then a probability of 2^{-2} of obtaining a valid message. We can repeat the previous procedure for 4 different chaining values, and expect one to give a valid padded last block. Thus we have a time complexity of 2^{33} of finding a (second) preimage with a correct padding.

7 Conclusion

In this paper, we have presented new results in several directions on the security of Keccak: A distinguisher on 4 rounds, preimage and collision attacks on 2 rounds, and near collisions on 3 rounds. These results apply to the 256 and to the 224 bits versions. First, these results concern the reduced round hash function rather than building blocks like the internal permutation or the compression function as considered in previous work. Next all our results are practical and have been implemented. The only known previous external cryptanalysis on the reduced round hash function setting was a marginally better than generic theoretical preimage attack. The number of rounds reached is far from the total, and the results do not present a threat against the whole hash function, but we believe they will contribute to a better understanding of the security of the Keccak hash function.

References

1. Bernstein, D.J.: Second preimages for 6 (7? (8??)) rounds of Keccak? NIST mailing list (2010), http://ehash.iaik.tugraz.at/uploads/6/65/NIST-mailing-list_Bernstein-Daemen.txt
2. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak reference. Submission to NIST (Round 3) (2011), http://keccak.noekeon.org/Keccak-reference-3.0.pdf
3. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak SHA-3 submission. Submission to NIST (Round 3) (2011), http://keccak.noekeon.org/Keccak-submission-3.pdf
4. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: On the indifferentiability of the sponge construction. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 4965, pp. 181–197. Springer (2008)

5. Biham, E., Chen, R.: Near-Collisions of SHA-0. In: Advances in Crypology: CRYPTO 2004. Lecture Notes in Computer Science, vol. 3152, pp. 290–305. Springer (2004)
6. Boura, C., Canteaut, A., Cannière, C.D.: Higher-order differential properties of Keccak and *Luffa*. In: FSE. Lecture Notes in Computer Science, vol. 6733. Springer (2011)
7. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional differential cryptanalysis of NLFSR-based cryptosystems. In: ASIACRYPT 2010. Lecture Notes in Computer Science, vol. 6477, pp. 130–145. Springer (2010)
8. Morawiecki, P., Srebrny, M.: A SAT-based preimage analysis of reduced KECCAK hash functions. Cryptology ePrint Archive, Report 2010/285 (2010), `http://eprint.iacr.org/2010/285.pdf`
9. Naya-Plasencia, M.: How to Improve Rebound Attacks. In: Advances in Crypology: CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 188–205. Springer (2011)
10. Rechberger, C., Rijmen, V.: On authentication with hmac and non-random properties. In: Dietrich, S., Dhamija, R. (eds.) Financial Cryptography and Data Security. Lecture Notes in Computer Science, vol. 4886, pp. 119–133. Springer Berlin / Heidelberg (2007)
11. Turan, M.S., Uyan, E.: Near-collisions for the reduced round versions of some second round SHA-3 compression functions using hill climbing. In: INDOCRYPT. Lecture Notes in Computer Science, vol. 6498, pp. 131–143. Springer (2010)
12. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: CRYPTO. Lecture Notes in Computer Science, vol. 3621, pp. 17–36. Springer (2005)
13. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 3494, pp. 19–35. Springer (2005)

A Information for the Distinguisher

A.1 Free Bits Which are in the 1088 Bit Message

In Table 3 we listed all the free bits from Section 3.

A.2 Bits in the Hash Output With a Bias

In the following, we give the list of bits in the hash with a bias of absolute value 2^{-1}. The numbering corresponds to the bit position in the 256-bit hash:

- For all $i \in \{22, 119, 126, 128, 138, 169, 205\}$ we have $\epsilon_i^{\mathcal{H}^{(4)}} = 2^{-1}$.
- For all $i \in \{56, 63, 98, 127, 149, 161, 162, 176, 195, 232, 252\}$ we have $\epsilon_i^{\mathcal{H}^{(4)}} = -2^{-1}$.

B Hash Function Near-Collision Example for 3 Rounds

In this section we give an example for a near collision after two rounds. The two input messages collide in 234 out of 256 bits.

- input 1:

Table 3. Free bits.

$(x=4, y=0, z=0)$	$(x=4, y=2, z=0)$	$(x=4, y=1, z=4)$	$(x=4, y=2, z=4)$
$(x=2, y=0, z=15)$	$(x=2, y=1, z=15)$	$(x=2, y=2, z=15)$	$(x=1, y=0, z=20)$
$(x=1, y=2, z=20)$	$(x=1, y=3, z=20)$	$(x=1, y=0, z=23)$	$(x=1, y=1, z=23)$
$(x=1, y=2, z=23)$	$(x=0, y=0, z=24)$	$(x=4, y=0, z=24)$	$(x=0, y=1, z=24)$
$(x=4, y=1, z=24)$	$(x=0, y=2, z=24)$	$(x=4, y=2, z=24)$	$(x=0, y=3, z=24)$
$(x=0, y=0, z=27)$	$(x=0, y=1, z=27)$	$(x=0, y=2, z=27)$	$(x=3, y=0, z=28)$
$(x=3, y=1, z=28)$	$(x=3, y=2, z=28)$	$(x=2, y=0, z=30)$	$(x=2, y=1, z=30)$
$(x=2, y=2, z=30)$	$(x=0, y=1, z=31)$	$(x=0, y=2, z=31)$	$(x=0, y=3, z=31)$
$(x=4, y=0, z=34)$	$(x=4, y=1, z=34)$	$(x=3, y=0, z=35)$	$(x=4, y=0, z=35)$
$(x=3, y=1, z=35)$	$(x=4, y=1, z=35)$	$(x=3, y=2, z=35)$	$(x=4, y=2, z=35)$
$(x=2, y=0, z=36)$	$(x=2, y=1, z=36)$	$(x=2, y=2, z=36)$	$(x=3, y=0, z=37)$
$(x=3, y=1, z=37)$	$(x=3, y=2, z=37)$	$(x=1, y=0, z=39)$	$(x=1, y=1, z=39)$
$(x=2, y=0, z=42)$	$(x=2, y=1, z=42)$	$(x=3, y=0, z=43)$	$(x=3, y=1, z=43)$
$(x=1, y=0, z=44)$	$(x=1, y=1, z=44)$	$(x=1, y=2, z=44)$	$(x=1, y=3, z=44)$
$(x=0, y=1, z=47)$	$(x=0, y=2, z=47)$	$(x=0, y=3, z=47)$	$(x=0, y=0, z=54)$
$(x=2, y=0, z=54)$	$(x=0, y=1, z=54)$	$(x=2, y=1, z=54)$	$(x=0, y=2, z=54)$
$(x=2, y=2, z=54)$	$(x=1, y=0, z=56)$	$(x=1, y=1, z=56)$	$(x=1, y=2, z=56)$
$(x=1, y=3, z=56)$	$(x=1, y=0, z=57)$	$(x=1, y=1, z=57)$	$(x=1, y=2, z=57)$
$(x=1, y=3, z=57)$	$(x=0, y=0, z=60)$	$(x=0, y=1, z=60)$	$(x=0, y=2, z=60)$
$(x=0, y=3, z=60)$	$(x=0, y=0, z=61)$	$(x=0, y=1, z=61)$	$(x=0, y=2, z=61)$
$(x=0, y=3, z=61)$			

```
0x09c2d45d03bae701a767c1b756e7e594c38ad4c618efc11dc32289
  31bb698feb072e3f9a6e9e8b414942e18102755b2e2faf545ac717
  402e12ac5f93ce54484955a870311867e2095b981797d778ee2e7e
  e3fa8fcb24e650ada1c4a07344f79ab8672027c502b240dda77eb9
  39c89134e778718ab86e39f75524e8a200c025ac0bdce3b246ddc5
```

- input 2:

```
0x09c2d45d03bae701a767c1b756e7e594c38ad4c618efc11dc32289
  31bb698feb072e3fda6e9e8bc14942e18102755b2e2faf545ac717
  402e12ac5f93ce54484955a870311867e2095b9817d7d778ee2e7e
  e3fa8fcb24e650ada1c4a07344f69ab8672027c502b240dda77eb9
  39c89134e778718ab86e39775524e8a200c025ac0bdce3b246ddc4
```

we get the following difference in the hash value:

- output difference:

```
0x000020000100800920000000814080102009300009000008000810
  0000000001
```

C Hash Function Collision Example for 2 Rounds

For the two inputs:

- input 1:

```
0x22458a902041831f3f7ffe62c58b16d1a3468df8f1e2c524499327
  22458b17dab468d1983061c2c2850a1543860d1a860d1b37112346
  8ca850a041a040800151a3468c52a44890993366ccca94295267cf
  9f3e448811220c18306093264c988a142953be7cf9f22e5dbb7792
  244992a4499327b265ca951f3e7cf81a3468d053a74f9fe8d1a244
```

- input 2:

```
0x22458a902041821f3f7ffe62c58b06d1a34685f8f1e2c524499327
  22458b17dab468d1983061c2e2850a1543860d1b860d1b37112346
  9ca850a041a040800151a3468c52a44890993366ccca94295267cf
  9f3e448811220c18306093264c988a14295bbe7cf9f22e5dbb7792
  244992a4499327b265ca953f3e7cf81a3468d053a74f9fe8d1a244
```

we get the same output, namely:

- output:

```
0xfa5f041d6152cc9b8f0747aa9f66b1be5365164ff14578436665f4
  e828f1ea76
```

D Preimage for 2 Rounds of the 256-bit Hash Function

We chose as target output the all zero hash. The following message block, which has the correct final padding, fed to the 2 rounds version of the 256-bit Keccak hash function, outputs a hash value of all zeros.

- input inclusive the padding:

```
0x8ed557e2ad30f0ee00000000000000000000000000000000e9a08c
  fd1399f7ad00000000000000000000000000000000cb1dc30d0ad7
  92214a0bb03f6aedcb230000000000000000a4a82caa2dfa4d388e
  d557e2ad30f0ee00000000000000004a0bb03f6aedcb23e9a08cfd
  1399f7ada4a82caa2dfa4d380000000000000000cb1dc30d0ad79221
```