Preimage attacks on the round-reduced Keccak with the aid of differential cryptanalysis

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

[1] Section of Informatics, University of Commerce, Kielce, Poland

[2] Department of Computing, Macquarie University, Australia

[3] Institute of Computer Science, Polish Academy of Sciences, Poland

Abstract. In this paper we use differential cryptanalysis to attack the winner of the SHA-3 competition, namely Keccak hash function. Despite more than 6 years of intensive cryptanalysis there have been known only two preimage attacks which reach 3 (or slightly more) rounds. Our 3-round preimage attack improves the complexity of those two existing attacks and it is obtained with a different technique. We also show the partial preimage attack on the 4-round Keccak, exploiting two properties of the linear step of the Keccak-f permutation.

Keywords: preimage attack, Keccak, differential cryptanalysis, SHA-3

1 Introduction

In 2007, the U.S. National Institute of Standards and Technology (NIST) announced a public contest aiming at the selection of a new standard for a cryptographic hash function. The main motivation behind starting the contest were the security flaws identified in the SHA-1 standard in 2005. Similarities between SHA-1 and the most recent standard SHA-2 were worrisome and NIST decided that a new, stronger hash function would be needed. Overall, 51 functions were submitted to the first round of the contest. In July 2009 out of the submitted functions, 14 were selected to the second round. At the end of 2010, the five finalists were announced and eventually in October 2012 the winner has been selected. The new SHA-3 standard will be Keccak hash function [4].

In this paper we mount a preimage attack against the round-reduced Keccak. We use differential cryptanalysis where an attacker tries to deduce some useful information by tracking an evolution of the xor difference between two messages.

We focus on the variants proposed as SHA-3 candidates. Our main result is a preimage attack on the 3-round Keccak (applicable to Keccak-512 and Keccak-384) and the 4-round partial preimage attack against Keccak with default settings proposed by the designers. The complexity of the attack on 3 rounds is lower than the best current attacks for this number of rounds. Table 1 shows the comparison. Our attacks take advantage of 3-round differential paths which work with probability 1 or very close to 1. Additionally, we exploit two properties of the linear step θ.

Among the attacks on the Keccak hash function, the most rounds were reached by Bernstein in his 8-round preimage attack [2]. However, the attack is much slower than the parallel exhaustive search and it is inherently memory-intensive. With the aid of differential analysis, Naya-Plasencia et al. mounted the preimage and collision attacks on the 2-round Keccak [12]. In [10] the same result (2-round preimage and 2-round collision attacks) were obtained through the SAT-based attacks. Very recently rotational cryptanalysis led to a 4-round preimage attack [11] and some ideas from that work are present in our analysis. The most successful collision attacks (up to 4 rounds) was given in [6, 7].

The distinguisher of Keccak's permutation with the highest number of rounds is the zero-sum distinguisher proposed in [1] and later improved in [5, 8]. However, the complexity of these

Table 1. Best known preimage attacks on the Keccak variants as SHA-3 candidates. The number in the column 'Variant' denotes the hash length.

Rounds	Variant	Time	Memory	Reference
6/7/8	512	$2^{506}/2^{507}/2^{511.5}$	$2^{176}/2^{320}/2^{508}$	[2]
4	512	$2^{505.3}$	2^{61}	[2]
3	512	$2^{505.2}$	2^{61}	[2]
4	512	2^{506}	negligible	[11]
3	512	2^{506}	negligible	[11]
3	512	$2^{503.7}$	negligible	Section 4

distinguishers is very high. For example, the zero-sum distinguisher for all 24 rounds has the complexity of 2^{1579}. A differential analysis of KECCAK's internal permutation, given in [9], leads to distinguishers up to 8 rounds with complexity of $2^{491.47}$

2 Keccak

In this section we provide a description of Keccak to the extent necessary for understanding the attack described in the paper. For a complete specification, we refer the interested reader to the original specification [4].

Keccak uses the sponge construction and hence is a member of the sponge function family [3]. It can be used as a hash function but also can be applied for generating an infinite bit stream, making it suitable as a stream cipher or a pseudorandom bit generator. In this paper we focus on the sponge construction for cryptographic hashing. Keccak has two main parameters r and c, which are called bitrate and capacity, respectively. The sum of those two parameters makes the state size, which Keccak operates on. In the SHA-3 proposal the state size is 1600 bits. Different values for bitrate and capacity give trade-offs between speed and security. A higher bitrate gives a faster function at the expense of a lower security. Keccak follows the sponge two-phase processing.

The initial 1600-bit state is filled with 0's. In the first phase (also called the absorbing phase), the state is processed by consecutive applications of the permutation Keccak-f. The absorbing phase is finished when all message blocks have been processed. In the second phase (also called the squeezing phase), the first r bits of the state are returned as part of the output bits, interleaved with applications of the permutation Keccak-f. The squeezing phase is finished after the desired length of the output digest has been produced.

For the variants proposed as the SHA-3 standard, the value of the parameter c is equal to the hash length multiplied by 2. For example, the SHA-3 candidate with 512-bit hash length is Keccak with $c = 1024$ and $r = 576$ $(r + c = 1600)$. In this paper we denote variants proposed as the SHA-3 candidates by Keccak-512, Keccak-384, Keccak-256, and Keccak-224. (The number at the end of the name specifies the hash length.)

Keccak can also operate on smaller states but through the whole paper we always refer to the default variant with the 1600-bit state. The state can be visualised as an array of 5×5 lanes, where each lane is a 64-bit string. The state size determines the number of rounds in Keccak-f function. For the default 1600-bit state, there are 24 rounds. All rounds are the same except for constants, which are different for each round. In the paper when referring to the first half of a round we mean linear steps θ, ρ, π and when referring to the second half a round we mean χ and ι.

Below there is a pseudo-code of a single round. In the latter part of the paper, we often refer to the algorithm steps (denoted by Greek letters) described in the following pseudo-code.

```
Round(A,RC) {

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

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

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

$\chi$ step                                      forall (x,y) in (0...4,0...4)
A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),

$\iota$ step
A[0,0] = A[0,0] xor RC

return A   }
```

All the operations on the indices shown in the pseudo-code are done modulo 5. `A` denotes the complete permutation state array and `A[x,y]` denotes a particular lane in that state. `B[x,y]`, `C[x]`, `D[x]` are 64-bit intermediate variables. The constants `r[x,y]` are the rotation offsets, while `RC` are the round constants. `rot(W,m)` is the usual bitwise rotation operation, moving bit at position i into position $i+m$ in the lane `W` ($i+m$ are done modulo 64 – note that 64 is the lane size for the default variant of Keccak). θ is a linear operation that provides diffusion to the state. ρ is a permutation that mixes bits of a lane using rotation and π permutes lanes. The only non-linear operation is χ, which can be treated as a layer of 5-bit Sboxes. Finally, ι xores the round constant with the first lane.

Properties of θ exploited in attacks

In our attacks we exploit two properties of θ. We use the notion of a 'column parity kernel' (or CP-kernel) introduced by the Keccak's designers [4]. A 1600-bit state difference is in the CP-kernel if all its columns have even parity. Such states are fixed points for θ and thus θ does not increase the Hamming weight in such cases. When we start a differential characteristics from a low Hamming weight CP-kernel it allows us to stay in a very low Hamming weight difference state through the first round. This property is used in all attacks presented in the paper. The second property exploited in the attacks is a simple observation that all 5 bits in a given column are either left unchanged or all 5 are flipped. So, for example, even if an attacker does not know all the differences in a given column, still she can say something on selected bit differences after applying θ. This property is used in the 4-round preimage attack.

3 2.5-round differential distinguisher

Before we present an actual preimage attack let us first describe 2.5-round differential distinguisher. Our aim here is to distinguish the Keccak-f[1600] permutation from a random permutation.

In a basic variant of differential cryptanalysis we trace xor differences between pairs of plaintexts. For a random permutation we assume that a difference between any output bits follows the binomial distribution $\mathcal{B}(t, s)$, where t is a number of trials and s is a probability of success and is equal to 0.5. The mean for the binomial distribution equals $s \cdot t$ and the standard deviation $\sigma = \sqrt{(1-s)s \cdot t}$. Now we will be able to distinguish the round-reduced Keccak-f permutation from a random permutation if, for example, starting from a fixed input difference we are going to get always zero output differences on some carefully selected bits.

In our analysis we trace xor differences between corresponding bits from two states. A bit with the difference 1 is called 'active', and with the difference 0 is called 'inactive'. When a difference is unknown and we do not put any constraints on it, it is called 'unknown'.

Usually when building a differential path, a cryptanalyst assumes some probabilistic transitions on certain non-linear parts (such as sbox layers). Here our approach is different as for our 3-round attack we need paths, which work with probability 1. Thus once an initial difference between plaintexts is set, we let differences propagate without assuming any probabilistic transitions across the non-linear step in the permutation.

A differential path starts with 4 active bits such as two of them are in a column and the other two active bits are in other column. We choose this setting for two reasons:

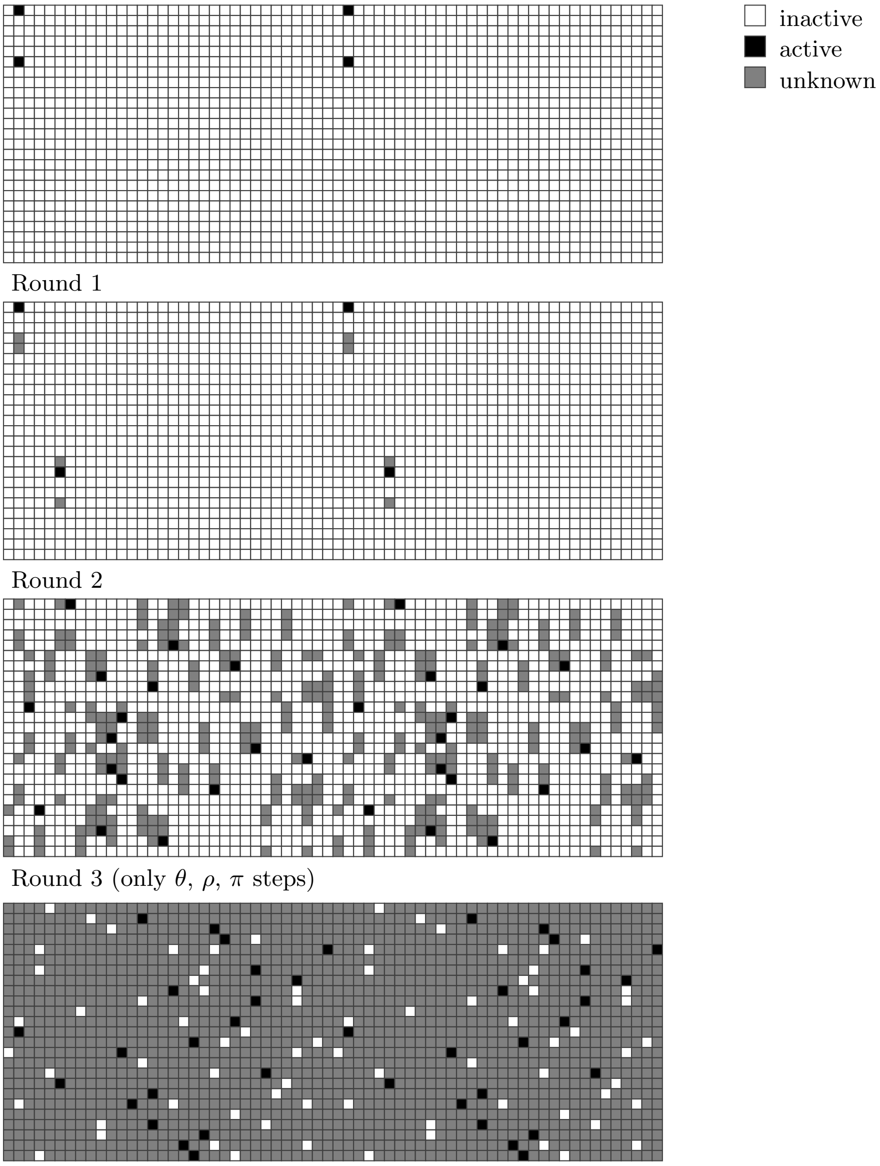
- The difference state is in the column parity kernel and hence after first θ there are still only 4 active bits.
- For our preimage attack we need a family of differential paths and we can construct other paths by simply changing the columns with active bits and obtain a sufficiently big number of differential paths. If we used a path with only two active bits (placed in the same column), we would have fewer combinations and consequently the attack would be slower. The details on this are given in next section where we analyze our preimage attack.

Since θ, ρ and π are all linear, calculation how the differences change after these steps is straightforward. In the non-linear χ each bit of the state is calculated through a simple non-linear function taking 3 bits as its input ($y = x_1 \oplus \bar{x_2} x_3$). If x_2 or x_3 is an active or unknown bit, an output difference is unknown. This is because the difference depends on actual values and as we trace differences only, we can determine output differences with some probability. Hence such bit is marked as 'unknown'. ι, the last step of the permutation, xores the first lane with a round constant and does not affect our analysis in any way.

Figure 1 illustrates a differential path created according to the description given above. For the sake of diagram readability instead of 5x5 matrix of lanes there are 25 rows, each representing a single lane (64 bits). We stress that the path happens with probability 1. In the 1st round, θ does not spread differences because the initial xor difference of states is in the column-parity kernel. After χ in the 1st round unknown bits start to appear. They are spread through the state in the 2nd round and at the end of the 2nd round their number is substantial. The 3rd round's θ makes nearly all the state filled with unknown bits. Luckily, there are still some active or inactive bits and they will be used in our preimage attack. Eventually χ in the 3rd round changes these useful bits to unknown bits.

There are more such differential paths (distinguishers), which behave in similar way to the one shown in Figure 1. The initial four active bits can be placed in different columns (keeping in mind that they should be in the column parity kernel). A family of such distinguishers lets us mount a preimage attack.

Fig. 1. 2.5-round differential path with probability 1



4 3-round preimage attack

Here we describe the 3-round preimage attack on Keccak-512 but the attack works in exactly the same way for other Keccak's variants proposed as the SHA-3 standard. The goal of our attack is to find a message for a given 512-bit hash h. We assume that a message has 512 bits and is properly padded, fitting into a single block of 576 bits. Note that for Keccak-512 bitrate $r = 576$ and capacity $c = 1024$. With 512 unknown message bits we can expect that among 2^{512} possible messages there is, on the average, one with the given hash h.

In our attack we want to find a pair of messages which follow a certain differential path and one of these messages is a preimage we are looking for. We will show that the workload for this task is below the exhaustive search of 2^{512} values.

Before we describe the actual attack, let us focus on the precomputation phase. In this phase we first generate 1024 differential distinguishers of the form as shown in the previous section.

We could try to find a higher number of distinguishers (higher than 1024) but it does not make the attack faster. We will give details on this later in the section. The distinguishers are for 2.5 rounds and each one has a different input xor difference, that is two pairs of bits, where each pair is placed in a different column. We consider only first 64 columns and as we choose 2 columns out of 64, there are $\binom{64}{2} = 2016$ possible choices. In our attack, we choose 1024 distinguishers that for the chosen pattern of input active bits, they produce at least 12 active or inactive bits on their outputs. Actually, this condition is fulfilled for nearly all 2016 possible choices so basically we can choose at random. Once the distinguishers are generated we keep in memory a list of positions of these 12 active or inactive bits. Each distinguisher has its own list of input and output active and inactive bits. To remember a position one needs 9 bits of memory and one extra bit for marking whether it is active or inactive bit. Then totally we need $1024 \cdot 12 \cdot 10 = 120$ kBytes which is negligible amount.

In our attack we try to guess 512 unknown preimage bits. Certainly, the probability of correct guess is 2^{-512}. However, a probability that a guessed message along with the preimage forms a right pair for any of the 1024 differential paths is $1024/2^{512} = 2^{-502}$. (By a right pair here we mean a pair whose input and output patterns of active/inactive bits follow the pattern for one of the distinguishers) Then after 2^{502} guesses, one would hit a message that together with the preimage (we are looking for) follows the differential path of one of the 1024 distinguishers. By examining active/inactive patterns of a difference output, we can determine (with a low cost) which exactly differential path (out of 1024 considered) it follows. Then obtaining a preimage is straightforward — xor the guessed message with the input difference of this path. This is the idea behind our attack.

For a description of the attack we use the following notation:

h - a given 512-bit hash being a result of the 3-round Keccak-512.

H - first 320 bits of the state after 2.5 rounds of Keccak-512 calculated from h. Note that if a whole row (5 bits) is known, we can invert these bits through ι and χ from an originally given hash h. It is because χ operates on the rows independently and can be inverted on a row-by-row basis. For a 512-bit hash, there are 64 whole rows then $64 \cdot 5 = 320$ bits can be inverted through ι and χ.

$H^{'}$ - a hash generated by an attacker by running the 2.5-round Keccak-512 on a guessed message.

$s^{n}_{(x,y,z)}$ - status (active or inactive) of a bit after 2.5 rounds in the n-th distinguisher (out of 1024 considered). A bit of the Keccak state is determined by its position (x, y, z).

For the code readability we denote 3-round Keccak-512 variant by Keccak$_3$-512 or generally a reduced k-round variant by Keccak$_k$. Here is the attack given in the pseudo-code:

Algorithm 1: Preimage attack on the 3-round Keccak-512
Input: A given hash h
Output: A message m such that h=Keccak$_3$-512(m)

```
$preimageFound$ := **false**;
**while** ($preimageFound$ = **false**) **do**
  1. guess a 512-bit message $m$;
  2. $H^{'}$ = Keccak$_{2.5}$-512($m$); (run the 2.5-round Keccak-512 on the guessed message $m$)
  3. **for** $n := 0$ **to** $n < 1024$ **do**
```

```
(a) candidate := true;
(b) for all 12 coordinate triplets $(x, y, z)$ being on the list created in precomputation do
        if $(H_{(x,y,z)} \oplus H^{'}_{(x,y,z)} = 0)$ and ($s^n_{(x,y,z)}$ is active) then candidate := false;
        if $(H_{(x,y,z)} \oplus H^{'}_{(x,y,z)} = 1)$ and ($s^n_{(x,y,z)}$ is inactive) then candidate := false;
(c) if (candidate = true) then xor the guessed message $m$ with the input difference of
    the $n$-th distinguisher and run the 3-round KECCAK-512 on it to check whether it is
    a preimage of a given hash. If ($h$=Keccak$_3$-512($m$)) then preimageFound := true;
```

Inside the most inner loop the attacker computes a xor difference between certain bits of H and $H^{'}$. If the result does not comply with the bit status (active/inactive) defined by the differential path n, then the guessed message is rejected. (It is the point in the pseudo-code where a variable *candidate* becomes false.)

It may happen that a guessed message is not rejected although it generates a wrong pair. This is a false positive. The are many such false positives and their number can be calculated as follows. For each distinguisher there is a list of 12 active or inactive bits. (Those lists were created in precomputation.) The probability that we hit a false positive for which all 12 bits are consistent with the 12 bits on the list is 2^{-12}. Hence there will be around $2^{512}/2^{12} = 2^{500}$ false positives to check.

Let us analyse the computational complexity of the most inner loop. For each distinguisher n there are 12 bits to check. Checking one bit can be implemented with 2 bitwise XOR operations (first xor between $H_{(x,y,z)}$ and $H^{'}_{(x,y,z)}$, and then xor with a status $s^n_{x,y,z}$). Then combining all these 12 results into a single bit answer (a message is rejected or not) can be implemented with 11 ORs. Therefore a total number of bitwise operations for the most inner loop is $12*3+11 = 47$. The 3-round Keccak requires 24192 bitwise operations then if we want express the most inner loop workload in Keccak calls we have $47/24192 = 0.001942791 \simeq 2^{-9}$ Keccak calls.

Now, we calculate a total workload for the attack. In the main loop we run Keccak on the guessed messages 2^{502} times. Plus the cost of operations in the most inner loop: $2^{502} \cdot 1024 \cdot 2^{-9} = 2^{503}$. Finally, we would have to check 2^{500} false positive candidates. So the total workload of the attack is $2^{502} + 2^{500} + 2^{503} \simeq 2^{503.7}$ Keccak-512 calls.

Discussion of the chosen parameters of the attack

The presented attack works also for other Keccak variants proposed as the SHA-3 standard beating the exhaustive search by the factor of $2^{8.3}$. Please note that the workload for the attack depends on a number of differential distinguishers in a limited way. The reason is that the most costly factor (all bitwise operations in the most inner loop) does not depend on a number of differential distinguishers. In the attack we use 2^{10} distinguishers, but let us suppose we can find a bigger number, say 2^{13}, with the same properties. Then the cost of the most costly factor would be: $(2^{512}/2^{13}) \cdot 2^{13} \cdot 2^{-9} = 2^{503}$ — the same value as in the presented attack. The main loop is indeed run fewer times $(2^{512}/2^{13})$ but we pay for it by checking more distinguishers.

What influences the most costly factor and the total workload of the attack is the number of bits checked in the most inner loop. Let us denote this number by x. The optimal value of x (giving the lowest complexity of the attack) is 12 and this number is chosen for the attack. It was determined in the following way. As explained above, the most inner loop can be implemented with $3x$ XORs + $(x - 1)$ ORs. The 3-round Keccak requires 24192 bitwise operations. Hence if we want to express the most inner loop workload in Keccak calls we have $(3x + (x - 1))/24191$. The total workload is then

$$w(x) = 2^{502} + 2^{512} \cdot (3x + (x - 1))/24191 + 2^{512-x}$$

and the function $w(x)$ has a minimum at $x = 12$ (when only integers are considered).

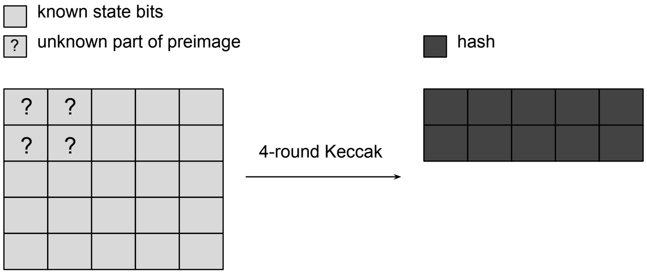
The attack was inspired by the rotational cryptanalysis on Keccak [11] where a similar approach of mounting a preimage attack is given. However, our work here uses differential cryptanalysis (instead of rotational one) which leads to better results for the 3-round variant and makes the method more applicable and robust. If the Keccak's initial state were filled with some random value (and not all 0's), the rotational attack would not have been working. Also, if the round constants had a higher Hamming weight, it would severely limit the rotational preimage attack. These two modifications do not influence or limit the power of our attack in any way.

Our approach should be applicable to functions (or its reduced variants) with a low diffusion. When some hash bits after a certain number of rounds do not depend on all message bits, then a family of distinguishers can be constructed and mounting the attack should be possible.

4.1 Partial preimage attack on the 4-round Keccak

A strong cryptographic hash function should have, among many other desired properties, partial preimage resistance. With a given k-bit preimage and l known bits of this preimage, the effort to find the rest of $k - l$ bits should be 2^{k-l} hash function calls. Here we want to show a partial preimage attack on the 4-round Keccak. We attack the Keccak variant with the default parameters proposed by the Keccak designers [4]. These are: r=1024 and c=576. The Keccak designers denote this default variant by Keccak[] and we use also this notation. We set a hash length to 640 bits and assume that a preimage has also 640 bits, out of which 256 bits are unknown. Figure 2 shows positions of the unknown bits in the state.

Fig. 2. Position of bits in the 4-round partial preimage attack. Each square represents a lane.

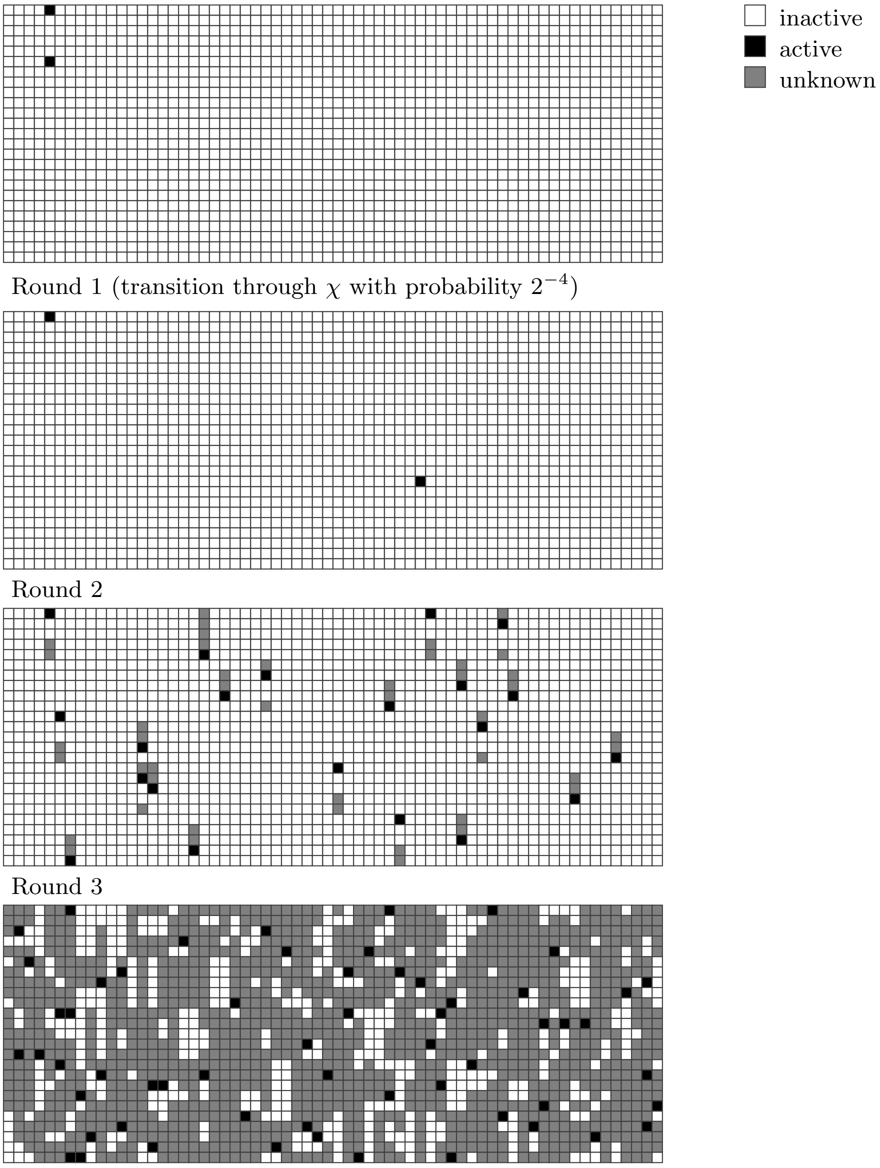


The distinguishers used in the 3-round preimage attack cover only 2.5 rounds so it is not possible to directly extend the attack to more rounds. We reach 4 rounds by exploiting the property of θ and allowing probabilistic transitions in the first round.

We use differential paths which start from two active bits placed in the same column. After the first non-linear χ the path can diverge into 16 possible paths. We assume that after the non-linear χ there are maximally three active bits. This condition is fulfilled by 5 paths (out of 16). Thus the probability that a pair of messages follows one of the paths we are interested in is $5/16 \simeq 2^{-1.7}$. The other paths, with more active bits, are not useful for our 4-round attack as unknown bits start dominating the state too early.

For subsequent steps of the paths we let differences propagate without assuming any probabilistic transitions across the non-linear steps. Such paths cover 3 rounds with many active or inactive bits at the end of the 3rd round. This is still not enough to mount the attack for 4

Fig. 3. 3-round differential path with probability 2^{-4}



Round 1 (transition through χ with probability 2^{-4})

Round 2

Round 3

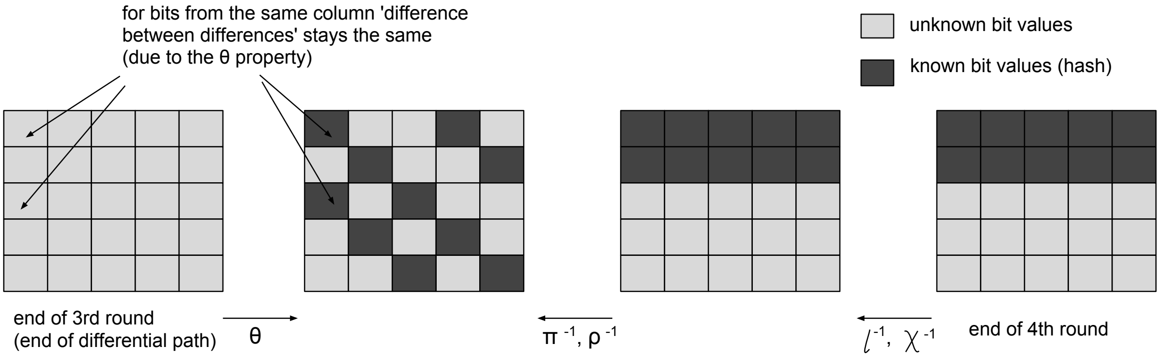
rounds. What makes it possible is the property of θ — the effect of θ on a given column is that either all 5 bits remain unchanged or all 5 bits are flipped. The same applies for xor differences.

As explained in the previous section, we can go back from a given hash (the end of the 4th round in this case) up to θ in the 4th round. We can not go back through θ as we would need to know the state bits which are not accessible for the attacker. However, at the end of the 3rd round, for the characteristics described above, some information can be deduced on an xor difference between bits from the same column. For example, let two bits from the same column be active. Because of the θ property, these two bits after θ would be either both active or both inactive. If a pair follows a given characteristics, it is not possible that other combination appears (like one bit active and the other inactive). Informally speaking, for bits from the same column the xor difference between differences stays the same before and after θ. This property lets us check whether we hit a right pair and in consequence we can mount the attack for 4 rounds.

Now let us give the details for our partial preimage attack. The description and analysis of the attack is similar to the 3-round preimage attack but we need to take into account the probabilistic transitions and the θ property we exploit. We consider only the paths which start from 2 active bits that are in the same column while the rest of bits are inactive. Figure 3 shows an example path. For 256-bit unknown part of the state (located as shown in Figure 2), there are 128 such starting points (input state differences of the paths). As explained above, for each starting point there are 5 differential paths which are useful for the attack. We refer to it as a family of subpaths (for a given starting point). Therefore, totally we consider $128 \cdot 5 = 640$ differential paths.

In the attack the adversary knows 640-bit hash — first 10 lanes of the state after 4 rounds. When we go back from the hash to θ in the 4th round, 10 lanes of bits are known but their places in the state have changed. There are five pairs of lanes ([0,0] and [0,2]; [1,1] and [1,3]; [2,2] and [2,4]; [3,0] and [3,3]; [4,1] and [4,4]) having the same x index (We refer to a given lane in the state by $[x, y]$.) In other words, these pairs share the same columns. As our differential paths cover only 3 rounds, we do not know the status of a particular bit after θ in the 4th round. However, what we do know is that certain pairs of bits (from the same column) for which we know the status (active or inactive) at the end of the 3rd round would behave according to the θ property we have described. Let us call these pairs of bits 'useful pairs'. Figure 4 illustrates the idea. Please note that if a given hash would be shorter, say 512 bits, then we would be able to invert only first 64 rows (5 top lanes) which is not enough to exploit the θ property.

Fig. 4. Combining a differential path with known values from a given hash. Each square represents a lane.


for bits from the same column 'difference between differences' stays the same (due to the θ property)
unknown bit values
known bit values (hash)
end of 3rd round (end of differential path)
θ
π -1, ρ -1
ι -1, χ -1
end of 4th round

In the precomputation phase, for each of the paths we consider, we mark these useful pairs of bits. There are at least 8 useful pairs for each path.

Here is our 4-round partial preimage attack given in the pseudo-code. Symbols h, H, $H^{'}$ and $s^n_{(x,y,z)}$ have the following meaning.

h - a given 640-bit hash being a result of the 4-round Keccak[].

H - 640 bits calculated by inverting h through ι and χ in the 4th round

$H^{'}$ - a hash generated by an attacker by running the 3.5-round Keccak[] on a guessed message.

$s^n_{(x,y,z)}$ - status (active or inactive) of a bit after 3 rounds in the n-th distinguisher (out of 128 considered). An active status is encoded by 1 and inactive by 0.

```
Algorithm 2: Partial preimage attack on the 4-round Keccak[]
Input: A 640-bit message $m$ with 256 unknown bits and a 640-bit hash $h$
Output: A complete message $m$ such that $h$=Keccak$_4$[]($m$)

preimageFound := false;
while (preimageFound = false) do
  1. guess 256 unknown bits of message $m$;
  2. $H^{'}$ = Keccak$_{3.5}$[]($m$); (run the 3.5-round Keccak[] on the guessed message $m$)
  3. for $n := 0$ to $n < 128$ do
     (a) candidate :=true;
     (b) check whether the guessed message $m$ follows any of the subpaths (starting from the
         $n$-th input difference). If ($m$ does not follow any of the subpaths) then break
         else record the subpath;
     (c) for all 8 useful pairs of bits of the recorded subpath (pairs of bits were determined
         in precomputation and have coordinates $(x, y, z)$ and $(x, y', z)$) do
           if $(H_{(x,y,z)} \oplus H^{'}_{(x,y,z)} \oplus H_{(x,y',z)} \oplus H^{'}_{(x,y',z)}) \neq (s^{n}_{(x,y,z)} \oplus s^{n}_{(x,y',z)})$
           then candidate:=false;
     (d) if (candidate=true) then xor the guessed message $m$ with an input difference of
         the $n$-th distinguisher and run the 4-round KECCAK[] on it to check whether it is a
         preimage of a given hash. If ($h$=Keccak$_4$[]($m$)) then preimageFound := true;
```

The following analysis is nearly the same as for the 3-round preimage attack. Probability that a guessed message along with the actual preimage form one of the 128 input state differences is $128/2^{256} = 2^{-249}$. Then the probability that after the first χ we are still on one of the paths of interest is $2^{-1.7}$. Therefore after $2^{249+1.7} = 2^{250.7}$ guesses we should hit a message which along with an unknown preimage gives a right pair. So, the main loop of the attack iterates $2^{250.7}$ times.

As in the attack on 3 rounds there will be many false positives to check. A probability that we hit on a candidate which, for all of its 8 useful pairs of bits, is not detected as a false candidate is 2^{-8}. Hence there will be around $2^{256}/2^{8} = 2^{248}$ false positive candidates to check.

Let us count how many bitwise operations (besides running Keccak[]) is done in the attack. The cost of Step 3(b) is calculated as follows. The guessed message m follows one of the subpaths provided that 4 bits after the linear steps in the 1st round have certain values. To calculate these 4 bits we do $4 \cdot 11 = 44$ XORs. Another 4 XORs determine the exact subpath the message may follow (or it says it would not follow any of them). So, Step 3(b) requires 48 XORs. Step 3(b) is repeated for each of 128 starting input differences, then in one iteration of the main loop of the attack it costs $48 \cdot 128 = 6144$ XORs. The 4-round Keccak[] has 32256 bitwise operations then if we want to express the calculated number in Keccak's calls we have $6144/32256 = 0.19047619 \simeq 2^{-2.4}$. The main loop of the attack is iterated $2^{250.7}$ times, so the total cost of Step 3(b) is equivalent to $2^{250.7} \cdot 2^{-2.4} = 2^{248.3}$ 4-round Keccak[] calls.

Additionally, there is a cost of the most inner loop (Step 3(c)). The condition there requires 5 XORs and is computed for each of 8 pairs. Combining all these 8 results into a single bit answer (a message is rejected or not) can be implemented with 7 ORs. So, totally $(5 \cdot 8 + 7) = 47$ bitwise operations for one Step 3(c). Step 3(c) is calculated only if Step 3(b) does not break the loop (which happens with probability $11/16 \simeq 2^{-0.54}$). The main loop of the attack is iterated $2^{250.7}$ times, so the total cost of Step 3(c) is $2^{250.7-0.54} \cdot 47 \simeq 2^{255}$ bitwise operations which is equivalent to $2^{255}/32256 \simeq 2^{240}$ 4-round Keccak[] calls.

Summing up, the total workload of the attack is: $2^{250.7}$ (the number of guessed messages) + 2^{248} (checking false positives) + $2^{248.3}$ (Step(b)) + 2^{240} (Step(c)) $\simeq 2^{251.12}$ 4-round Keccak[] calls. This is better than the exhaustive search by the factor of $2^{4.88}$.

Trying to mount the 4-round preimage attack (instead of partial preimage) raises the following obstacle. With 640-bit hash (as explained earlier a hash can not be shorter to exploit the θ property) we would guess 640-bit preimage. Therefore the bitrate r should have at least 640 bits and consequently the capacity c is maximally $1600 - 640 = 960$ bits long. The claimed security for such settings is $2^{c/2} = 2^{480}$. Therefore our attack would not be an actual attack as the complexity would be much higher than the claimed security.

5 Conclusion

In this paper we have presented a preimage attack on the 3-round Keccak-512 and a partial preimage attack on the variant of 4-round Keccak hash function. Despite the substantial research effort during the SHA-3 competition and afterwards, there have been only two preimage attacks which reach 3 rounds or more [2, 11]. We have introduced a new one and our 3-round attack has a lower complexity than the two other 3-round attacks and use negligible amount of memory. Our approach is more applicable than the rotational attack and should be useful particularly for attacking hash functions (or their reduced variants) with low diffusion. We also have shown that θ has some other property (besides the CP-kernel) which can be exploited in a partial preimage attack. It allows us to reach 4 rounds.

References

1. Aumasson, J.P., Meier, W.: Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi. Tech. rep., NIST mailing list (2009)
2. 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`
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic sponges, `http://sponge.noekeon.org`
4. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak sponge function family main document, `http://keccak.noekeon.org/Keccak-main-2.1.pdf`
5. Boura, C., Canteaut, A.: Zero-Sum Distinguishers for Iterated Permutations and Application to Keccak-f and Hamsi-256. In: Biryukov, A., Gong, G., Stinson, D. (eds.) Selected Areas in Cryptography, Lecture Notes in Computer Science, vol. 6544, pp. 1–17. Springer Berlin Heidelberg (2011)
6. Dinur, I., Dunkelman, O., Shamir, A.: Collision attacks on up to 5 rounds of sha-3 using generalized internal differentials. Cryptology ePrint Archive, Report 2012/672 (2012), http://eprint.iacr.org/
7. Dinur, I., Dunkelman, O., Shamir, A.: New Attacks on Keccak-224 and Keccak-256. In: Canteaut, A. (ed.) Fast Software Encryption, Lecture Notes in Computer Science, vol. 7549, pp. 442–461. Springer Berlin Heidelberg (2012)
8. Duan, M., Lai, X.: Improved zero-sum distinguisher for full round Keccak-f permutation. Chinese Science Bulletin 57, 694–697 (2012)
9. Duc, A., Guo, J., Peyrin, T., Wei, L.: Unaligned Rebound Attack - Application to Keccak. Cryptology ePrint Archive, Report 2011/420 (2011)
10. Homsirikamol, E., Morawiecki, P., Rogawski, M., Srebrny, M.: Security margin evaluation of SHA-3 contest finalists through SAT-based attacks. In: 11th Int. Conf. on Information Systems and Industrial Management. LNCS, vol. 7564. Springer Berlin Heidelberg (2012)
11. Morawiecki, P., Pieprzyk, J., Srebrny, M.: Rotational cryptanalysis of round-reduced Keccak. In: Fast Software Encryption. LNCS, Springer (2013)
12. Naya-Plasencia, M., Röck, A., Meier, W.: Practical analysis of reduced-round keccak. In: Bernstein, D., Chatterjee, S. (eds.) Progress in Cryptology INDOCRYPT 2011, Lecture Notes in Computer Science, vol. 7107, pp. 236–254. Springer Berlin Heidelberg (2011)