1st and 2nd Preimage Attacks on 7, 8 and 9 Rounds of Keccak-224,256,384,512

Donghoon Chang[1], Arnab Kumar[2], Paweł Morawiecki[3] and Somitra Kumar Sanadhya[1]
{donghoon@iiitd.ac.in, arnabkumar225@nsitonline.in, pawel.morawiecki@gmail.com, somitra@iiitd.ac.in}

[1]Indraprastha Institute of Information Technology - Delhi, New Delhi, India
[2]Netaji Subhas Institute Of Technology, New Delhi, India.
[3] Polish Academy of Sciences, Institute of Computer Science, Poland.

Abstract. In this work, we show a general method to find preimages for a cryptographic hash function H which is composed of two parts as $H = H_1 \circ H_2$. We use the ideas of Dinur and Shamir (FSE 2011) and Bernstein (SHA-3 mailing list) for the H_1 part, invert a part of H_2 and connect the two using a table lookup to find preimages for H. The suggested approach works whenever the internal constituents of H_1 have low algebraic degree and H_2 has low diffusion.
We show an application of this technique on round-reduced variants of the Keccak hash function, the winner of the SHA-3 competition. Using our approach, we show new 1st and 2nd preimage attacks on 7-round Keccak-224, 8-round Keccak-256/384 and 9-round Keccak-512. The previous best preimage attack was on 8 round Keccak. Our attacks take advantage of the low degree of Keccak permutation, such that the output of each round of Keccak can be represented as an equation of degree 2 in terms of its input bits. Since the draft SHA-3 standard (FIPS 202) and Keccak only differ in padding scheme, our approach extends to round-reduced SHA-3 variants as well. The complexities of the attack increase as the number of rounds attacked increase. None of the attacks threaten the security of Keccak as the attack complexities are already close to brute force by the time we cross 9 rounds of Keccak. In fact, this work shows the limits of polynomial enumeration method based preimage attacks against Keccak.

Keywords: Keccak, preimage attack, second preimage attack

1 Introduction

Keccak, designed by Guido Bertoni, Joan Daemen, Michael Peetërs, and Gilles Van Assche, is a cryptographic hash function built upon an earlier design, Radiogatún [3], by the same authors. It was selected as the winner of the NIST Secure Hash Algorithm 3 competition [7] in Oct. 2012. All the Keccak versions submitted to the SHA-3 competition have an input state size of 1600 bits, and an output size $o \in \{224, 256, 384, 512\}$ bits. SHA-3 uses the Sponge construction [2], where the message blocks are first XOR'ed with the initial bits of the state, and then permuted through 24 rounds of a non-linear yet invertible round function.

Despite being in the scrutiny of cryptographers for the past few years, Keccak has remained immune to any serious threat. Previous results on Keccak's internal permutation include zero-sum distinguisher presented in [1], and later improved in several papers [5, 6, 13]. Practical attacks on Keccak include Naya-Plasencia, Rock and Meier's work on Keccak-224 and Keccak-256 [18], which include a preimage attack on 2 rounds, as well as collisions on 2 rounds and near-collisions on 3 rounds. Morawiecki et al. [17] used SAT solvers to show a preimage attack on 3 round Keccak, and then used rotational cryptanalysis [16] to show a preimage attack on 4 round Keccak. Recently, Dinur, Dunkelman and Shamir [9] have developed practical collision finding attacks, extending up to 4 rounds. Finally, in the NIST Keccak mailing list, Bernstein [12] described a second preimage attack extending to Keccak-512 reduced to 8 rounds. Bernstein's attack is marginally faster than exhaustive search and uses a huge amount of memory.

Since this work concerns the study of preimage and second preimage attacks on Keccak, we list all the previous attacks of this type and our results obtained in the current work in Table 1.

The paper is organized as follows. We briefly describe the Keccak design in § 2, explain the terminology used in § 3 and present an overview of previous polynomial enumeration based preimage attacks on Hamsi and Keccak in § 4. We then describe the polynomial enumeration algorithm in details in § 5. We then present the idea of using lookup tables to gain some extra rounds in § 6 in combination with the polynomial enumeration. This algorithm is used to perform a preimage attack on Keccak. Our attacks are described in § 7. These attacks are further improved based partial last-round inversion in § 8. This is in contrast to the full last-round inversion in § 7. We conclude in § 9.

Table 1. A summary of the results on the Preimage and Second Preimage attacks on Keccak. *Complexity* is determined in terms of number of calls of the compression function and the *Improvement Factor* denotes the ratio of the complexity of generic (exhaustive search) attack and the attack under consideration. We compare our best results with earlier reported optimal results [12, 8, 14] on reduced round Keccak-512. Similar improvements can be obtained for other versions of Keccak.

Version	**Reference**	**No. of Rounds**	**Type of attack**	**Time Complexity**	**Memory Complexity**	**Improvement Factor**
Keccak-256	[18]	2	Preimage	2^{33}		$\mathbf{2^{223}}$
Keccak-512	[17]	3	Preimage	2^{506}		**64**
Keccak-512	[16]	4	Preimage	2^{506}		**64**
Keccak-512	[12, 8]	6	2nd Preimage	2^{506}	2^{176}	**50**
	[12, 8, 14]	7	”	2^{507}	2^{320}	**37**
	[12, 8, 14]	8	”	$2^{511.4}$	2^{508}	**1.44**
	This work, § 7	6	Preimage/ 2nd Preimage	$2^{509.19}$	$2^{98.91}$	**7.01**
	This work, § 7	7	”	$2^{509.39}$	$2^{172.52}$	**6.13**
	This work, § 7	8	”	$2^{509.73}$	$2^{315.29}$	**4.81**
Keccak-224	This work, § 8	7	”	$2^{218.11}$	$2^{180.12}$	**58.66**
Keccak-256	This work, § 8	8	”	$2^{255.64}$	$2^{254.03}$	**1.29**
Keccak-384	This work, § 8	8	”	$2^{378.74}$	$2^{324.06}$	**38.36**
Keccak-512	This work, § 8	6	”	$2^{505.58}$	$2^{104.23}$	**85.70**
	This work, § 8	7	”	$2^{506.11}$	$2^{180.12}$	**59.34**
	This work, § 8	8	”	$2^{506.74}$	$2^{324.07}$	**38.36**
	This work, § 8	9	”	$2^{511.70}$	$2^{510.02}$	**1.23**

2 Description Of Keccak

The Keccak family consists of four cryptographic hash functions, called Keccak-224, Keccak-256, Keccak-384, and Keccak-512. The numerical suffix indicates the fixed length of the digest, e.g., Keccak-256 produces 256-bit digests etc.

Keccak uses the Sponge Construction [4]. The version of Keccak used for SHA-3 operates on an internal state of $t = 1600$ bits, which is divided into bitrate η and capacity c. The message block is first XOR'ed with the bitrate, after which the full state of t bits is rearranged by a permutation which is operated 24 times. This operation is called a "round". Processing of the message in this construction is divided into two phases: the absorption phase and the squeezing phase. The internal state of the Sponge construction consumes the message words and modifies itself via the Keccak permutation in the absorption phase. Once the message is completely consumed, a part of the state is then output in successive rounds of the squeezing phase. Each round R of the permutation is obtained by the composition of five mappings as follows. $R = \theta \circ \rho \circ \pi \circ \chi \circ \iota$. Note that the first three permutations are linear while the last two are non-linear. A detailed description of all the mappings involved in Keccak is available in the Keccak Reference 3.1, available on the official Keccak website [4]. We refer the reader to [3] for further details on the round function.

A message is padded by the rule $\text{pad}(m) = m||10^*1$ in Keccak, with the asterisk denoting the number of zeroes required to get the size of the padded message to be a multiple of bitrate. The four SHA-3 hash functions do have a few differences from the instances of Keccak that were proposed for the Keccak competition. In particular, two additional bits(01) are appended to the messages, before the standard Keccak padding is applied. This is done in order to differentiate the SHA-3 hash functions from the SHA-3 extendable-output functions (XOF) [19]. Thus, at least 4 bits of padding is applied to a message before being processed by the draft SHA-3 as against at least two bits of padding in the case of Keccak.

3 Terminology

We first explain the terminology and the basic setting used in this work.

We consider a hash function $H : \{0,1\}^* \to \{0,1\}^o$. The ith bit of the output of H can be considered as a function f_i mapping $\{0,1\}^n \to \{0,1\}$, where these n bits are a subset of the input block. We consider only 1 block of message, where the message size after padding is m bits. Note that $n \leq m$, i.e., we do not consider *all* the m bits of input to f_i but only some specific n bits. This will become clear later when the attack is described.

The output of f_i on a specific n bit input x can be written as a sum of monomials in the input bits x_1, x_2, ..., x_n. We assume that the maximum degree of the monomials in the description of f_i is at most d.

Let α be the set of all sets of cardinality up to d constructed out of a universe of size n, with elements represented as $1, 2, \ldots, n$. This set can be represented in the following two equivalent ways.

$$\begin{aligned}\alpha &= \{\phi, \{1\}, \ldots, \{1,2,3\}, \cdots, \{n-d+1, \cdots, n\}\}, \\ &= \{A : A \subseteq \{1,2,3,\ldots,n\}; 0 \le |A| \le d\}.\end{aligned}$$

As illustrated, A is an element of α. If A is non-empty then we define it as $A = \{a_1, a_2, \ldots, a_k\}$, where $1 \le k \le d$ such that $a_1 < a_2 < a_3 \ldots < a_k$. This set A of cardinality k is converted to a set S_A, which is an n bit string. The set S_A corresponds to an n-bit string specifying whether or not a specific element is present in A. This enables it to be used for computation of the function f_i and later H. We define S_A as $S_A = (s_1, s_2, \ldots, s_\sigma, \ldots, s_n)$ where $s_\sigma = 1$ if $\sigma \in A$ otherwise $s_\sigma = 0$ for $1 \le \sigma \le n$. We extend this definition by introducing a new term $S_{A,p}$ for $a_p \in A$. Let $A = (a_1, a_2, \ldots, a_k)$. We define $A_p = A - \{a_p\}$. The term $S_{A,p}$ can be defined as equal to $(s'_1, s'_2, \ldots, s'_n)$ where $s'_i = 1$ if $i \in A_p$, otherwise $s'_i = 0$.

We write f_i with n bit input $x = x_1 x_2 \ldots x_n$ as the sum of monomials in its input bits. As already mentioned, we assume that the maximum degree of the monomials is d. Thus, we can express f_i as follows.

$$\begin{aligned}f_i &= f_i(x_1, x_2, \ldots, x_i, \ldots, x_n) \\ &= C_\phi + C_{\{1\}}x_1 + C_{\{2\}}x_2 + \ldots + C_{\{1,2\}}x_1x_2 + \ldots + C_{\{(n-d+1),\ldots,n\}} \prod_{t\in\{(n-d+1),\ldots,n\}} x_t\end{aligned} \tag{1}$$

4 Previous preimage attacks based on Polynomial enumeration

Our attack on Keccak hinges on a "polynomial enumeration algorithm" which has already been used by Dinur and Shamir [11] in the case of Hamsi-256, and by Bernstein [12] in the case of Keccak-512.

4.1 Polynomial enumeration

Polynomial enumeration is an algorithm to efficiently interpolate and evaluate a polynomial of degree d, given some inputs and corresponding outputs (or a subset of output bits). We describe this algorithm next.

1. Compute the number of monomials in n variables having degree less than or equal to d. The number of monomials obtained is exactly
$$|\alpha| = \sum_{j=0}^{d} \binom{n}{j}. \tag{2}$$
2. For every element A of α having max size/degree d, compute the coefficient C_A associated with it. According to Dinur and Shamir [11], complexity of computing all these coefficients is $\sum_{l=0}^{d} 2^{l-1} \times \binom{n}{l}$ bit operations. However, a separate computation of the same, suggested by Bernstein in his letter to the Keccak Team [12], showed that this computation could have been achieved at the cost of $\sum_{l=0}^{d} l \times \binom{n}{l} + T \times \sum_{l=0}^{d} \binom{n}{l}$ bit operations, where the hash function computation takes T bit operations. Since this process is repeated for b bits of the output, the time complexity associated with this step is $b \times \sum_{l=0}^{d} l \times \binom{n}{l} + T \times \sum_{l=0}^{d} \binom{n}{l}$ bit operations. Finally, we allocate a coefficent static array C of size 2^n, which is initialized with all zeros, and copy each coefficient C_A of the polynomial into $C[S_A]$, where S_A is the n-bit string as defined in the previous section. However, he did not describe such algorithm in a formal way. Moreoever, we need more time and memory complexity to describe such algorithm in a formal way. Bernstein's idea is explained in details as Algorithm 1 with detailed explanation of increased time and memory complexity in § 5.1 and a detailed example of this process is provided in the Appendix.
3. Use "Moebius transform" [12] to obtain the truth table static array of size 2^n from the coefficent static array C given in Step 2. This step is highly memory intensive as 2^n bit memory is required for computing each bit of the output (refer to § 6). Thus the memory complexity for b bits is $b \times 2^n$. The time complexity for each bit is $n \times 2^{n-1}$. For b bits, the time complexity of this step is
$$b \times n \times 2^{n-1}. \tag{3}$$
4. All the above steps of the polynomial enumeration algorithm are performed 2^{o-n} times, where o is the output size. Such a number of iterations ensure that almost all the input bits are considered. It is to be noted that all possible values of the input are not to be considered. Checking the output for all possibilities of o bits in the input ensures that the probability of finding a preimage/second preimage following this process is approximately equal to 1.

We direct the reader to similar attacks on round-reduced Hamsi-256 by Dinur and Shamir [11] and on round-reduced Keccak-512 by Bernstein [12].

5 Speeding-up Polynomial enumeration

Bernstein [12], in an informal note, suggested storing the partial sums and reusing them to speed-up the polynomial enumeration algorithm. We show an algorithm based on Bernstein's suggestion.

5.1 Algorithm for Computing the Coefficient Static Array C of Size 2^n for a n-bit input Boolean Function

The algorithm for computing a coefficient array of a given boolean function f in the polynomial enumeration is presented in Algorithm 1.

Algorithm 1: Computing the Coefficient Static Array of a Boolean Function

```
Input: Boolean function $f$ with $n$-bit input and having algebraic degree at most $d$
Result: Coeffient static array $C$ of size $2^n$, which is initialized with all zeros in the beginning
1  begin
2      $l$=0;
3      while $l \leq d$ do
4          for $A \in \alpha$ AND $|A| = l$ do
5              $y$=0;
6              $i$=0;
7              $y$=$f(S_A)$;
8              $\mathrm{Sum}_0[S_A] = y$;
9              while $i < l$ do
10                 $y = y \oplus \mathrm{Sum}_i[S_{A,i+1}]$;
11                 $i$=$i$+1;
12                 $\mathrm{Sum}_i[S_A] = y$;
13             $C[S_A] = y$, where $C_A$ is also same as $C[S_A]$;
14         $l$=$l$+1;
```

Time Complexity of the Algorithm to compute they coefficient static array of a boolean function f : The time complexity of the algorithm is dependent on the three loops running in the program as follows.

1. The outermost *while* loop ensures that the coefficients of all terms upto degree d are evaluated.
2. The *for* loop in the middle ensures that coefficients of all terms of a specific degree l, such that $l \leq d$ are evaluated. This loop runs $\binom{n}{l}$ times. Further, the value for all $f(S_A)$ is to be computed as well. Let the time complexity of computing f be T bit operations. Then the time required to perform this step is $T \times \binom{n}{l}$. This step needs to repeat for $l = 0$ to d.
3. The innermost *while* loop computes the coefficient for each and every term of the polynomial. As observed in the algorithm, it involves l bit XORs, or l bit operations. In addition, the step incrementing i costs about 2 bit-wise XOR operations, on average. The reason for this is as follows. If the least significant bit (lsb) of i is 0, then the increment affects only the lsb. This happens with probability 1/2. However, with probability 1/2, the lsb is 1 and the increment causes a carry into higher order bits. With probability 1/4, this carry stops at the next bit else it propagates further. This analysis shows that the cost of incrementing i is $\sum_{i=1}^{d} \frac{i}{2^i}$ which is ≤ 2 for any d. In Steps 10 and 13, we need to take or update an element of Sum arrays. Note that each Sum array of size 2^n, whose element is 1-bit, are static so it will be constant time to take or update an element of a Sum array. Let w be the constant time to take or update an 1-bit element of a Sum array of size 2^n. In this paper, we assume that w=1. Note that Steps 7, 10, 11, and 12 occupy most of time complexity of Algorithm 1. Thus, the time complexity of Steps 7, 10, 11, and 12 is given by the following upper bound.

Time complexity $= (2w + 3) \times (\sum_{l=0}^{d} l \times \binom{n}{l}) + T \times \sum_{l=0}^{d} \binom{n}{l}$ bit operations.

Memory Complexity of the algorithm to compute the coefficient static array of f : The memory complexity of the algorithm is directly related to the 'Sum' terms being computed. Each 'Sum' term is defined as an array of size 2^n whose index size is n-bit. The inner *while* loop in Step 3 with loop counter being equal to l uses $\mathrm{Sum}_0[S_{A,1}]$ to $\mathrm{Sum}_{l-1}[S_{A,l}]$ for a given set A where $|A| = l$, i.e. the hamming-weight of the n-bit string S_A is l. Note that the n-bit strings $S_{A,1}$, $S_{A,2}$, …, $S_{A,l}$ have hamming-weight $(l-1)$. We have already stored values $\text{Sum}_0[S_B] = y$ for all $|B| = l-1$ in the previous step when the loop counter is $l-1$. We utilize these stored values to compute $\text{Sum}_0[]$, $\text{Sum}_1[]$, $\ldots$, $\text{Sum}_l[]$. Step 8 and Step 12 in Algorithm 1 require the $\text{Sum}_i[S_A]$ values to be stored. For any iteration, we need to store the current 'Sum' values and the previous 'Sum' values. That is, for a loop counter value l in Step 3, we need to use $\text{Sum}_0[S_{A_0}]$, $\text{Sum}_1[S_{A_1}]$, $\ldots$, $\text{Sum}_{l-1}[S_{A_{l-1}}]$ from the previous step for all A_i's such that $|A_i| = l-1$. Further, we need to store the current $\text{Sum}_i[S_A]$ values, for all $0 \leq i \leq l$ for all A such that $|A| = l$. In the next step of the while loop in step 3, when loop counter is $l+1$, we only need to use 'Sum' values from the previous step, and store the current 'Sum' values for the next step. That is, there is no need to save the values from the loop when the loop counter was $l-1$.

In order to store each element value of 'Sum' array, we need 1 bits of memory. There are $l+(l+1)$ 'Sum' arrays which are needed to be stored. Hence, the total memory complexity is $(2l+1)\times 2^n$ bits. This term attains its maximum value at $l = d$. Further, Step 13 of the algorithm reqiures to replace element values of the array C with the coefficients. This step takes 2^n additional memory to construct the array C. Hence, the maximum memory complexity of Algorithm 1 is given by

$$(2d+1)\times 2^n + 2^n.$$

Example of the Algorithm for computing coefficients : An example illustrating Algorithm 1 is provided in the Appendix.

6 General description of the attack

6.1 Algorithm

As explained earlier, the Polynomial enumeration method can be used to compute a polynomial relation between a subset of input and output bits. Such a relation can be used to compute some output bits for some subset of inputs. This, in turn, enables the attacker to determine the appropriate set of input bits for some known output bits. However, we introduce a slight modification to this method. Rather than running the polynomial enumeration based algorithms for the full hash function H and then calculate some specific b output bits, we consider the hash function as composed of two parts as $H = H_1 \circ H_2$.

Given a hash output, we first compute the positions of the q bit input to H_2 which affects b bits of final output. We store these input-output relations in a table. We expect 2^{q-b} relations to be stored. Then we use the polynomial enumeration algorithm to generate truthtable of q bits of output of H_1. Finally, we use a meet-in-the-middle step to connect the output of H_1 to the inputs of H_2. If we find a match then we have succeeded in finding a preimage for H. We repeat this process sufficient times to ensure that the probability of finding the preimage is almost 1. Note that the method of inverting one round (or a part of one round) of Keccak has already been used in [10, 15, 16, 18]. We introduce the idea of using a lookup table to perform the meet-in-the-middle step at the intersection of H_1 and H_2.

Our approach can be easily extended to get a second preimage attack, further details of which are mentioned in § 6.3. The attack algorithm is explained in details next. The complexity of each step is also mentioned in the description.

Initialization: This part of the algorithm constructs a lookup table which is used in finding the preimage later.

1. As already mentioned, we consider H as being composed of H_1 and H_2. The output of H is of length o bit. Let the message block be of size m bits, out of which some $(m-o)$ bits are fixed to any constant value (say all zeros for example) and n bits are variable. We assume that $m \geq o$ and $n \leq o$. The $(o-n)$ bits except the variable n bits are exhaustively searched in the preimage attack, as mentioned in Step 5 and 6 below. For example, we start from all zero bits in these $(o-n)$ bits and increment the bit pattern in these set of bits by 1 in each iteration of Step 5a and 5b.
2. We start by first determining the optimal set of b out of o output bits for H_2, such that the number of bits required to compute them (from the output of H_1) is minimum. Let the number of required output bits at the matching point be q.

The preimage attack: This part of the algorithm utilizes the table constructed in the previous steps. Given a fixed o-bit hash output, the following steps lead to a preimage.

3. For all possibilities of these q bits, we apply H_2 to obtain the final output at the specific b bit locations. We discard those inputs which do not provide the desired output in these b bit positions and save the rest in a lookup table. The expected number of q bit patterns remaining is $2^{(q-b)}$. Let the complexity of computing H_2 be T'', then this step has a cost of $2^q \times T''$ bit operations.

4. The lookup table is then sorted, using an algorithm such as merge sort, so that any specific element of q bits can be searched efficiently by using any appropriate search algorithm, e.g. binary search. This step has an effective complexity of $2^{q-b} \times \max\{q-b, 1\} \times (q)$ since the expected size of the table is $2^{(q-b)}$. This lookup table is maintained in a sorted order.
5. Repeat the following steps $2^{(o-n)}$ times by incrementing the middle $(o-n)$ bit pattern, or until we find a preimage.
   (a) At this point, we use polynomial enumeration algorithm to calculate q truth table arrays, where the size of each array is 2^n, corresponding the q output bits at the end of H_1 while starting the evaluation from n variable input bits at the beginning of the hash computation. Let the complexity of computing H_1 be T' bit operation, then the complexity of this step is $(T' \times \sum_{j=0}^{d} \binom{n}{j})) + ((2w+3) \times q \times \sum_{j=0}^{d} j \times \binom{n}{j})) + (q \times n \times 2^{n-1})$, where w is the time to take or update an 1-bit element of a static array of size 2^n. In this paper, we assume $w = 1$.
   We already have a sorted lookup table containing $2^{(q-b)}$ entries each of which is of q bit length. We match the q-bit entries in this table with the q truth table arrays and find all the indexes of the matched elements of the q truth table arrays, where each index of the matched elements is n-bit describing the actual values of the n variable input bits. The cost of this matching is $(\max\{q-b, 1\} \times 2^n \times q)$. This is by running binary search between the lookup table (which cosists of 2^{q-b} q-bit entries) and the q truth table arrays.
   (b) On average among 2^n possible messages, $2^{(n-b)}$ messages are matched for the specific b bits of the hash output. Recall that the matching is being done on partial hash output, while the complete hash is of o bit length. Once we have a match, we run the hash function H on $2^{(n-b)}$ messages and check if the output matches with the complete hash output of o bits. If this matches then we have found a preimage and we stop. Otherwise, we repeat Steps 5a and 5b by incrementing the value of the variable $(o-n)$ bit string. The complexity of running a hash function for 2^{n-b} messages is $T \times 2^{n-b}$ where T is the complexity of computing H.

We are repeating Step 5a and Step 5b 2^{o-n} times by incrementing the variable $(o-n)$ bit pattern. This ensures that sufficient number of input combinations are considered, such that the probability of finding the first and second preimage is very high.

As a result, the complexity of the algorithm described above is as follows.

$$\begin{aligned} & b \times 2^q \times T'' + 2^{q-b} \times (q-b) \times q + \\ & 2^{o-n} \times [(T' \times \sum_{j=0}^{d} \binom{n}{j})) + ((2w+3) \times q \times \sum_{j=0}^{d} j \times \binom{n}{j})) + (q \times n \times 2^{n-1})] + \\ & 2^{o-n} \times [(T \times 2^{n-b}) + (\max\{(q-b), 1\} \times 2^n \times q)], \end{aligned} \tag{4}$$

where o is the number of output bits of the hash function being analyzed and other variables have meaning as described earlier. Note that $T = T' + T''$.

Memory Complexity of the Attack: The memory complexity of our attack consists of two parts: (i) computing the Moebius transformation and (ii) preparing the lookup table. The memory Complexity associated with all other steps are negligible in comparison to these two. These costs can be analyzed as follows.

1. The Lookup Table: The lookup table consists of $2^{(q-b)}$ possible q-bit inputs for H_2, all of which produce b bits of the desired target hash value. As already mentioned, all entries of the lookup table are of q bits each. As a result, the memory complexity of the lookup table is $q \times 2^{q-b}$.
2. Algorithm 1: The memory complexity of this step is at most $(2d+1) \times 2^n + q \times 2^n$, where $(2d+1) \times 2^n$ is the memory size for Sum arrays and $q \times 2^n$ indicates the memory size for q coefficient arrays of size 2^n.
3. The Moebius transformation: we do not need more memory, because this transformation just transforms the coefficient arrays to the truth table arrays.

The total memory complexity is upper bounded by the following expression.

$$q \times 2^{q-b} + (2d+q+1) \times 2^n. \tag{5}$$

6.2 Choosing an Optimal Set of Output Bits to minimize q

Our choice of the number of output bits, and their respective positions are dependent on the number of output bits q at the end of H_1 which are necessary to compute the final output. An optimal choice of output bits would minimize q, thereby reducing the number of bits to be computed via the polynomial enumeration algorithm. The optimal choice depends on how have we divided H for the meet-in-the-middle step. The position of partitioning is the most crucial factor in determining both the time and the memory complexity of the attack.

It is not immediately clear how should one choose the specific location of the output bits for the attack to succeed with higher probability. It will depend on the specific hash function being attacked.

6.3 Second preimage attack

The attack described above is a preimage attack on any hash function. However, a second preimage attack similar to the preimage attack described above can be devised with small modofications. In the case of the second preimage attack, we have to ensure that the second preimage, or the second string of input bits for which we get the desired set of output bits, is not equal to the first preimage. As a result, to avoid such an occurence, the constant set of bits (m-o bits) have to be set such that they do not match with the corresponding set of input bits of the first preiamge. It is to be noted that m denotes the total number of input bits, whereas o signifies the number of output bits. Moreover, for practical purposes, the number of constant input bits to be verified different are taken to be much less than the number of output bits, ensuring that the complexity associated with the computation of the specific set of output bits does not add significantly to the overall cost of our attack.

Such a procedure ensures that the cost of the extra step of the preimage being different from the given string remains negligible in comparison to the overall attack cost. Thus, the cost of the second preimage attack is almost the same as that for the preimage attack.

7 Attack on Keccak and Result

7.1 Choosing Output bits to minimize q for Keccak

The 1600 bit state of Keccak is considered as an array of $64 \times 5 \times 5$ bits. Common representation for the structure of Keccak state is in the form of a cuboid, which is represented as an array of the form $a[0..4][0..4][0..63]$. As already mentioned, the choice of number and location of the output bits at the end of H_1 is crucial to the cost of the attack. We analyzed a number of possibilties for these bits. A detailed analysis of the permutations involved in the Keccak compression function, specifically χ, led us to conclude that all bits taken along a row would ensure the minimum value of q. Such a case arises due to the fact that the permutation χ involves displacement of bits along a given row. As a result, our choice of the output bits along the given row would lead them to being dependent upon themselves, thereby reducing q. Bits taken along other alignements produced a larger value of q since these bit locations can't cancel the effect of other Keccak permutations.

Once we have decided on the alignment of the output bits, we need to determine the number of bits, or in other words, the number of rows to be considered for output bits to be computed. This leads to experimentation with varying number of rows being chosen as the output bits and observing the value of the Improvement Factor for the same. Our analysis shows that the optimum balance between minimizing the value of q and the reduction in the number of Keccak compression function evaluations takes place when output bits are chosen along a single row. In all the calculations considered for this work, the number of output bits b is chosen to be 5, as each row consists of 5 bits, thereby leading to $q = 55$ bits being dependent on them. A pictorial representation of the 5 bits chosen for the purpose of this work is shown in Figure 1 in Appendix.

7.2 Attack on Keccak-512

6 Rounds: We consider the 5 output bits at the end of $r = 6$ rounds (as shown in Fig. 1 in the Appendix), which are dependent on 55 bits of the $(r-1)$th round. Since we are considering Keccak-512, there are 512 output bits. Therefore, we need not consider more than 512 bits of input (trying all 512 bits of the input would imply generic preimage attack).

Consider the number of monomials used in the attack given by Eqn. 2. This expression reduces slowly till $d \geq n/2$, but starts to decrease rapidly after $d < n/2$. Therefore, we need to ensure that $n > 2d$. Since the degree d of the expression which computes the 5 output bits is 32, we need $n > 64$. However, larger n has the counter effect of increasing the cost of the Moebius transformation step in Eqn. 3. Best trade-off is achieved between these two factors when the number of variables n are chosen to be 92. The cost of the attack in this case comes out to be as calculated below using Eqn. 4. Note that $n = 92$, $q = 55$, $b = 5$, $o = 512$, $d = 32$, $r = 6$ and bit complexity of computing H_1 is $5 \times 8064 = 40320$ (where H_1 is first 5 rounds of Keccak). The bit complexity of H_2 is 8064, where H_2 is the 6th round of Keccak. Therefore, the complete hash function $H = 48384$. The values of bit complexities for various rounds of Keccak are taken from [15], where the authors have mentioned the number of bit operations for 3-round Keccak to be 24192. As a result, the number of bit operations for each Keccak round is $24192/3 = 8064$ many bit operations. The complexity of our attack on 6-round Keccak-512 comes out to be $2^{509.19}$ hash function calls. This is 7.01 times faster than the generic preimage attack. The memory complexity of the attack is $2^{170.88}$ bits using Eqn. 5.

7 Rounds: In this case, $n = 165$, $q = 55$, $b = 5$, $o = 512$, $d = 64$, $r = 7$, cost of $H_1 = 48384$ bit operations and that of $H_2 = 8064$ bit operations. The cost of the attack is $2^{509.39}$, which is 6.13 times faster than exhaustive search. The memory complexity is $2^{172.52}$ bits.

8 Rounds: In this case, $n = 307$, $q = 55$, $b = 5$, $o = 512$, $d = 128$, $r = 8$, cost of $H_1 = 56448$ bit operations and that of H_2 is 8064 bit operations. The cost of the attack is $2^{509.73}$. This is 4.81 times faster than the exhaustive search attack. The memory complexity is $2^{315.29}$ bits.

8 Improving the attacks further

For most "good" hash functions, inverting few steps of the hash function will have high time/memory complexity. Therefore, to achieve optimal attck complexities, H_1 will cover $(r-1)$ rounds and H_2 will cover 1 round while attacking a hash function H of r rounds. That is, the meet-in-the-middle step will typically be executed one round before the final hash output. However, as attempted in some previous works [10, 15, 16, 18], a different approach works better in the case of Keccak.

The main idea is to do the partial inversion only for the last two permutation of the last rounds (i.e. only 0.4 rounds of inversion[1], covering χ and ι). Note that the first three permutations in the round function of Keccak are linear and therefore the algebraic degree, and hence the cost of performing polynomial enumeration step, for first 8.6 rounds (i.e. 8 rounds + the permutations θ, ρ and π) is the same as for the first 8 rounds. Then performing the meet-in-the-middle step eliminates the requirement of the look up table. Using this approach the attacks presented earlier can be improved further.

Note that $q \geq b$ due to the diffusion property of H_2. Once the algebraic degree has been fixed, the attack complexity can be made smaller by choosing smaller values of q and b. The optimal value for attacking the hash function H will be when when $q = b$ in Eqn. 4. The suggested approach allows to make $q = b$ for all the round-reduced versions of Keccak. For example, by choosing two rows ($b = 5$) for 9-round Keccak and using H_1 as 8.6 rounds, H_2 as 0.4 rounds we get a 9-round preimage attack on Keccak with lowest complexity.

We provide brief information about the attack on 9 round Keccak-512 (with other attacks being similar). In this case, H_1 = first 8.6 rounds of Keccak which can be computed in approximately 67712 bit operations and H_2 = last 0.4 rounds of Keccak = approximately 4864 bit operations. We take $n = 498$, $o = 512$, $b = q = 5$ and $d = 256$, then using Eqn. 4 and Eqn. 5, the time and memory complexities are $2^{511.70}$ and $2^{510.02}$ respectively. The improvement factor is 1.23. Results for 6, 7 and 8 round Keccak can be obtained in the same way and are shown in Table 1. Note that we take $b = q = 10$ for these attacks on round-reduced Keccak-224, 384, 512 and we take $b = q = 5$ on 8-round Keccak-256.

9 Conclusions and future work

In this work, we presented preimage and second preimage attacks on reduced-round versions of Keccak-224, 256, 384 and 512. By using a polynomial enumeration algorithm and a meet-in-the-middle strategy, we showed attacks on round-reduced variants of Keccak. Our approach is generic and can be applied to any hash function which has low algebraic degree. However, the approach is clearly limited by the degree associated with each round of the compression function being analyzed. For this reason, these attacks do not extend beyond 9 rounds of Keccak. In fact, even for the 9 round version of Keccak, the attacks are already close to brute force complexities in terms of number of compression function calls.

It remains an interesting open question to attack larger number of rounds of Keccak.

[1] Note that earlier works such as [15, 9] count these as 0.5 rounds. We are using a different convention.

Acknowledgements

We would like to thank Josef Pieprzyk for reading an earlier version of this draft and suggesting numerous editorial and technical suggestions. This version of the paper is significantly improved and made more readable due to his comments. Most importantly, the attacks can be improved significantly as shown in § 8 by ideas developed by him and a co-author of this paper (Paweł Morawiecki).

References

1. Jean-Philippe Aumasson and Dmitry Khovratovich. First Analysis of KECCAK. Available online, 2009. `http://131002.net/data/papers/AK09.pdf`.
2. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. Ecrypt Hash Workshop 2007, May 2007.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak specifications. Submission to NIST (Round 2), 2009. `http://keccak.noekeon.org/Keccak-specifications-2.pdf`.
4. Guido Bertoni. The Keccak sponge function family. `http://keccak.noekeon.org/index.html`. Accessed: 19 January, 2013.
5. Christina Boura and Anne Canteaut. Zero-sum distinguishers for iterated permutations and application to KECCAK-f and Hamsi-256. In *Proceedings of the 17th international conference on Selected areas in cryptography*, SAC'10, pages 1–17, Berlin, Heidelberg, 2011. Springer-Verlag.
6. Christina Boura, Anne Canteaut, and Christophe De Cannière. Higher-Order Differential Properties of KECCAK and *Luffa*. In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 252–269. Springer, 2011.
7. Chad Boutin. NIST Selects Winner of Secure Hash Algorithm (SHA-3) Competition. http://www.nist.gov/itl/csd/sha-100212.cfm. Accessed: 19 January, 2013.
8. Joan Daemen. Response to [12], date: 2 dec 2010. `http://ehash.iaik.tugraz.at/uploads/6/65/NIST-mailing-list_Bernstein-Daemen.txt`. Accessed: 20 July 2014.
9. Itai Dinur, Orr Dunkelman, and Adi Shamir. New attacks on Keccak-224 and Keccak-256. In *Proceedings of the 19th international conference on Fast Software Encryption*, FSE'12, pages 442–461, Berlin, Heidelberg, 2012. Springer-Verlag.
10. Itai Dinur, Paweł Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michal Straus. Practical Complexity Cube Attacks on Round-Reduced Keccak Sponge Function. IACR Cryptology ePrint archive, Report 2014/259, 2014. `http://eprint.iacr.org/2014/259.pdf`.
11. Itai Dinur and Adi Shamir. An improved algebraic attack on Hamsi-256. In *Proceedings of the 18th international conference on Fast software encryption*, FSE'11, pages 88–106, Berlin, Heidelberg, 2011. Springer-Verlag.
12. D.J.Bernstein. Second preimages for 6 (7? (8??)) rounds of Keccak?, Date: 27 Nov 2010. NIST mailing list `http://cr.yp.to/hash/keccak-20101127.txt`. Accessed: 19 January, 2013.
13. Ming Duan and Xuajia Lai. Improved zero-sum distinguisher for full round Keccak-f permutation. IACR Cryptology ePrint Archive, Report 2011/023, 2011. `http://eprint.iacr.org/2011/023`.
14. IAIK TU Graz. The SHA-3 Zoo, Keccak Cryptanalysis. `http://ehash.iaik.tugraz.at/wiki/Keccak#Cryptanalysis`. Accessed: 20 July 2014.
15. Paweł Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michal Straus. Preimage attacks on the round-reduced Keccak with the aid of differential cryptanalysis. IACR Cryptology ePrint archive, Report 2013/561, 2013. `http://eprint.iacr.org/2013/561.pdf`.
16. Paweł Morawiecki, Joseph Pieprzyk, and Marian Srebrny. Rotational cryptanalysis of round-reduced KECCAK. IACR Cryptology ePrint Archive, Report 2012/546, 2012. `http://eprint.iacr.org/2012/546.pdf`.
17. Paweł Morawiecki and Marian Srebrny. A SAT-based preimage analysis of reduced KECCAK hash functions. IACR Cryptology ePrint Archive, Report 2010/285, 2010. `http://eprint.iacr.org/2010/285.pdf`.
18. María Naya-Plasencia, Andrea Röck, and Willi Meier. Practical analysis of reduced-round Keccak. In *Proceedings of the 12th international conference on Cryptology in India*, INDOCRYPT'11, pages 236–254, Berlin, Heidelberg, 2011. Springer-Verlag.
19. National Institute of Standards and Technology (NIST). Draft FIPS 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, May 2014. Available at `http://csrc.nist.gov/groups/ST/hash/sha-3/sha-3_standard_fips202.html`.

A Example run of the algorithm for computing Coefficients

We show an example of how to apply the algorithm to compute coefficients associated with terms of a polynomial equation relating the input and output. We explain the same by providing an input-output table, and defining the polynomial function relating them. We then show how the algorithm yields the same polynomial function. Let us assume the following polynomial relation.

$$f(x_1, x_2, x_3) = x_1 + x_1x_2 + x_2x_3$$

Table 2. Input Bits x_1, x_2, x_3 and corresponding output produced

Input	Output produced
000	0
001	0
010	0
011	1
100	1
101	1
110	0
111	1

According to this polynomial function, the following input output combinations would occur.

To arrive at a polynomial function relating the input and output, we start with a general function composed of 3 variables in this case. The maximum degree of the monomials is taken to be 3.

$$f(x_1, x_2, x_3) = C_\phi + C_{\{1\}}x_1 + C_{\{2\}}x_2 + C_{\{3\}}x_3 + C_{\{1,2\}}x_1x_2 + C_{\{2,3\}}x_2x_3 \\ + C_{\{1,3\}}x_1x_3 + C_{\{1,2,3\}}x_1x_2x_3$$

We use algorithm 1 to obtain values of the coefficients associated with various terms of varying degrees. We start with degree 0 and move to terms of higher degrees.

A.1 Computing $\mathbf{C_\phi}$

1. For this computation, $A = \phi$.
2. According to the algorithm mentioned, one starts by computing the value of $f(000)$, which is then consequentially stored in a variable y . This value is then stored as $\text{Sum}_0(000)$, or the first partial sum used in computing C_ϕ. Since we do not enter into the loop, we assign the value of y as the value of C_ϕ. As observed in the table, the value of $f(000)$ is 0. Hence, value of C_ϕ is 0.

A.2 Computing $\mathbf{C_{\{1\}}}$

1. For this computation, $A = \{1\}$.
2. According to the algorithm, start by computing f(100), which is then consequentially stored in the variable y. This value is then stored as $\text{Sum}_0(100)$.
3. Now, we add $\text{Sum}_0[\text{S}_{A,1}]$ to y. $\text{Sum}_0[\text{S}_{A,1}]$ corresponds to $\text{Sum}_0[000]$, which has already been stored earlier. The value of y is then stored as $\text{Sum}_1[100]$. Since the program no longer enters into the loop, the updated value of y is now stored as $C_{\{1\}}$.
4. As seen in the last 2 steps, computation of C_1 would involve the following computation:

$$C_1 = f(100) + Sum_0[000].$$

Since f(100) is 1, as shown in the table, and $\text{Sum}_0[000]$ is 0, as saved earlier, the value of $C_{\{1\}}$ is 1.

A.3 Computing $\mathbf{C_{\{2\}}}$

1. For this computation, $A = \{2\}$
2. According to algorithm 1, we start by computing f(010), which is then consequentially stored in the variable y. This value is then stored as $\text{Sum}_0(010)$.
3. Now, we add $\text{Sum}_0[\text{S}_{A,1}]$ to y. $\text{Sum}_0[\text{S}_{A,1}]$ corresponds to $\text{Sum}_0[000]$, which has already been stored earlier. The value of y is then stored as $\text{Sum}_1[010]$. Since the program no longer enters into the loop, the updated value of y is now stored as $C_{\{2\}}$.
4. As seen in the last 2 steps, computation of $C_{\{2\}}$ would involve the following computation:

$$C_2 = f(010) + Sum_0[000].$$

Since f(010) is 0, as shown in the table, and $\text{Sum}_0[000]$ is 0, as saved earlier, the value of $C_{\{2\}}$ is 0.

A.4 Computing $\mathbf{C_{\{3\}}}$

1. For this computation, $A = \{3\}$.
2. According to algorithm 1, start by computing f(001), which is then consequentially stored in the variable y. This value is then stored as Sum_0[001].
3. Now, we add $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ to y. $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ corresponds to Sum_0[000], which has already been stored earlier. The value of y is then stored as Sum_1[001]. Since the program no longer enters into the loop, the updated value of y is now stored as $\mathrm{C}_{\{3\}}$.
4. As seen in the last 2 steps, computation of $\mathrm{C}_{\{3\}}$ would involve the following computation:

$$C_3 = f(001) + Sum_0[000].$$

Since f(001) is 1, as shown in the table, and Sum_0[000] is 0, as saved earlier, the value of C_3 is 0.

A.5 Computing $\mathbf{C_{\{1,2\}}}$

1. For this computation, $A = \{1, 2\}$.
2. Next, we start with f(110), which is then consequentially stored in the variable y. This value is then stored as Sum_0[110].
3. We then add $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ to y. $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ corresponds to Sum_0[010], which has already been stored earlier. The value of y is then stored as Sum_1[110].
4. Then add $\mathrm{Sum}_0[\mathrm{S}_{A,2}]$. It is to be noted that this expression correspond to Sum_1[100]. This sum is also stored as Sum_2[110]. Since the program no longer enters into the loop, the updated value of y is now stored as $\mathrm{C}_{\{1,2\}}$.
5. As seen in the last 3 steps, computation of $\mathrm{C}_{\{1,2\}}$ would involve the following computation:

$$C_{\{1,2\}} = f(110) + Sum_0[010] + Sum_1[100].$$

Since f(110) is 0, as shown in the table, Sum_0[010] is 0, as saved earlier, and the value of Sum_1[100] is 1, as saved earlier, the value of $\mathrm{C}_{\{1,2\}}$ is 1.

A.6 Computing $\mathbf{C_{\{2,3\}}}$

1. For this computation, A=$\{2, 3\}$
2. As per the algorithm, start with f(011), which is then consequentially stored in the variable y. This value is then stored as Sum_0[011].
3. We then add $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ to y. $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ corresponds to Sum_0[001], which has already been stored earlier. The value of y is then stored as Sum_1[011].
4. Then we add $\mathrm{Sum}_0[\mathrm{S}_{A,2}]$. It is to be noted that this expression correspond to Sum_1[010]. This sum is also stored as Sum_2[011]. Since the program no longer enters into the loop, the updated value of y is now stored as $\mathrm{C}_{\{2,3\}}$.
5. As seen in the last 3 steps, computation of $\mathrm{C}_{\{2,3\}}$ would involve the following computation:

$$C_{\{2,3\}} = f(011) + Sum_0[001] + Sum_1[010].$$

Since f(011) is 1, as shown in the table, Sum_0[001] is 0, as saved earlier, and the value of Sum_1[010] is 0, as saved earlier, the value of $\mathrm{C}_{\{2,3\}}$ is 1.

A.7 Computing $\mathbf{C_{\{1,3\}}}$

1. For this computation, $A = \{1, 3\}$.
2. According to algorithm 1, start with f(101), which is then consequentially stored in the variable y. This value is then stored as Sum_0[101].
3. We then add $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ to y. $\mathrm{Sum}_0[\mathrm{S}_{A,1}]$ corresponds to Sum_0[001], which has already been stored earlier. The value of y is then stored as Sum_1[101].
4. Then we add $\mathrm{Sum}_0[\mathrm{S}_{A,2}]$. It is to be noted that this expression correspond to Sum_1[100]. This sum is also stored as Sum_2[101]. Since the program no longer enters into the loop, the updated value of y is now stored as $\mathrm{C}_{\{1,3\}}$.
5. As seen in the last 3 steps, computation of $\mathrm{C}_{\{1,3\}}$ would involve the following computation:

$$C_{2,3} = f(101) + Sum_0[001] + Sum_1[100].$$

Since f(101) is 1, as shown in the table, Sum_0[001] is 0, as saved earlier, and the value of Sum_1[100] is 1, as saved earlier, the value of $\mathrm{C}_{\{1,2\}}$ is 0.

A.8 Computing $\mathbf{C_{\{1,2,3\}}}$

1. For this computation, $A = \{1, 2, 3\}$.
2. According to algorithm 1, start with f(111), which is then consequentially stored in the variable y. This value is then stored as Sum$_0$[111].
3. We then add Sum$_0$[S$_{A,1}$] to y. Sum$_0$[S$_{A,1}$] corresponds to Sum$_0$[011], which has already been stored earlier. The value of y is then stored as Sum$_1$[111].
4. Then we add Sum$_1$[S$_{A,2}$]. It is to be noted that this expression correspond to Sum$_1$[101]. This sum is also stored as Sum$_2$[111].
5. Then we add Sum$_2$[S$_{A,3}$]. It is to be noted that this expression correspond to Sum$_2$[110]. This sum is also stored as Sum$_3$[111].
6. As seen in the last 3 steps, computation of C$_{\{1,2,3\}}$ would involve the following computation:

$$C_{\{1,2,3\}} = f(111) + Sum_0[011] + Sum_1[101] + Sum_2[110].$$

Since f(111) is 1, as shown in the table, Sum$_0$[011] is 0, as saved earlier, value of Sum$_1$[101] is 1, as saved earlier, and the value of Sum$_2$[110] is 1, the value of C$_{\{1,2,3\}}$ is 0.

As a result, we could now summarize the value of all the coefficients in a table:

Table 3. Coefficients and their value

Input	Output produced
C_ϕ	0
C_1	1
C_2	0
C_3	0
$C_{1,2}$	1
$C_{2,3}$	1
$C_{1,3}$	0
$C_{1,2,3}$	0

Now, putting the values of the coefficients we obtained in the general equation described earlier.

$$f(x_1, x_2, x_3) = C_\phi + C_{\{1\}}x_1 + C_{\{2\}}x_2 + C_{\{3\}}x_3 + C_{\{1,2\}}x_1x_2 + C_{\{2,3\}}x_2x_3$$
$$+C_{\{1,3\}}x_1x_3 + C_{\{1,2,3\}}x_1x_2x_3$$

$$f(x_1, x_2, x_3) = 0 + 1x_1 + 0x_2 + 0x_3 + 1x_1x_2 + 1x_2x_3 + 0x_1x_3 + 0x_1x_2x_3$$

This equation finally gets reduced to the following.

$$f(x_1, x_2, x_3) = x_1 + x_1x_2 + x_2x_3 + x_1x_3$$

Hence, we use algorithm 1 to compute the boolean function used to compute output from the input.

A.9 Permutation and Dependencies on Bits of the Round $r - 1$

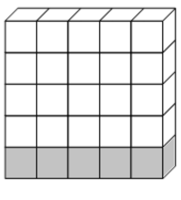

Fig. 1. Output bits chosen along the row of a slice.

Table 4. A summary of the bits required to compute 5 bits of Keccak output. To compute these bits, we invert the internal permutations of Keccak. The 5 output bits thus depend on 55 input bits only.

Permutation	**Bits at different positions of inversion of the Keccak permutations**
Initial Output bits	a[0..4][0][0]
After ι^{-1}	a[0..4][0][0]
After χ^{-1}	a[0..4][0][0]
After π^{-1}	a[0][0][0], a[1][1][0], a[2][2][0], a[3][3][0], a[4][4][0]
After ρ^{-1}	a[0][0][0], a[1][1][20], a[2][2][21], a[3][3][43], a[4][4][50]
After θ^{-1}	a[0][0][0], a[1][1][20], a[2][2][21], a[3][3][43], a[4][4][50], a[4][0..4][0], a[1][0..4][63], a[1][0..4][21], a[3][0..4][20], a[2][0..4][43], a[4][0..4][42], a[0][0..4][20], a[2][0..4][19], a[3][0..4][50], a[5][0..4][49]