Malicious SHA-3

Paweł Morawiecki

Institute of Computer Science, Polish Academy of Sciences, Poland

Abstract. In this paper, we investigate Keccak — the cryptographic hash function adopted as the SHA-3 standard. We propose a malicious variant of the function, where new round constants are introduced. We show that for such the variant, collision and preimage attacks are possible. We also identify a class of weak keys for malicious Keccak working in the MAC mode. Ideas presented in the paper were verified by implementing the attacks on the function with the 128-bit hash. Additionally, we show how the idea of malicious Keccak could be used in differential fault analysis against real Keccak working in the keyed mode such as the authenticated encryption mode.

Keywords: cryptanalysis, Keccak, SHA-3, differential fault analysis

1 Introduction

A malicious variant of a cryptographic primitive is one with a backdoor. An attacker who knows the backdoor can easily manipulate or even totally compromise the algorithm's security. A holy grail for all intelligence agencies is to have backdoors, which are extremely hard to detect and, at the same time, easy to use, widely applicable. Recent revelations by Edward Snowden have shown that the NSA has deliberately inserted a backdoor in the standardized pseudorandom number generator Dual_EC_DRBG [4]. This backdoor gives the knowledge of the internal state of the generator and consequently the attacker can predict future keystream bits. At the time of Snowden's revelations, NIST actually recommended Dual_EC_DRBG, so there are speculations that the NIST standards are manipulated by NSA, such as the NIST's recommended elliptic curve constants [14].

Malicious cryptography can be also an issue in commercial cryptography software or in some services relying on supposedly strong cryptographic algorithms. Imagine a multimedia online service, where a user buys and downloads videos and music. Clearly such a service should have a secure and reliable authentication mechanism. If the authentication is backdoored (e.g., a malicious hash function has been inserted), an anonymous worker can blackmail a company by showing an evidence of the system's weakness.

In the public domain, very few papers have been published regarding malicious cryptography. One of the first attempts was the cryptovirology project [18]. Another interesting try was to modify Sboxes of CAST and LOKI and hide linear relations inside [17]. Recently, a more formal treatment on malicious hashing was given, along with the malicious variant of SHA-1 [1]. (The constants of SHA-1 were modified in such a way that a collision attack is feasible.) In [11], a backdoored version of Streebog is presented. Streebog is a new Russian cryptographic hash standard (GOST R 34.11-2012) and its malicious variant has modified constants, which allows generating collisions, with an aid of differential cryptanalysis.

In this paper we focus on Keccak — the cryptographic hash function adopted as the SHA-3 standard [6]. We propose a new set of constants, which leads to collision and preimage attacks. Additionally, for malicious Keccak, we identify a class of weak keys. If a key belongs to the class, a forgery attack is possible. The idea behind new, malicious constants is to exploit the symmetric nature of the Keccak permutation. An inspiration comes from the previous attacks on Keccak such as rotational cryptanalysis [15] and internal differential cryptanalysis [9]. Both of these works take advantage of symmetry in the permutation and the low Hamming weights of the constants.

The paper is organised as follows. In Section 2, we describe Keccak and give the malicious constants. In Section 3, first we give the overview of internal differential cryptanalysis in context of the backdoored Keccak. Then, we explain how to attack malicious Keccak, particularly how to find collisions, preimages and mount a forgery attack on Keccak working in the MAC mode. In next section we show how the idea of malicious Keccak could be used in differential fault analysis. Before the paper is concluded, we discuss a different set of constants and its impact on the attack complexities.

2 Description of Keccak

Keccak is not a single algorithm but rather a family of algorithms called sponge functions [5]. A sponge function can be treated as a generalisation of the cryptographic hash function with infinite output. It can provide many functionalities, namely a hash function, a stream cipher, the keyed MAC or a pseudorandom bit generator. In this section we give a brief description of Keccak, necessary for understanding the malicious variant we propose and the attacks on the function. An interested reader finds all the details on Keccak in its original specification [6].

Keccak has the b-bit internal state, which is divided into two parts, that is the r-bit bitrate and the c-bit capacity ($r + c = b$). First, the state is filled with all 0's and a message is divided into r-bit blocks. Next, Keccak processes the message in two phases. The first phase is called the absorbing phase, where the r-bit message blocks are absorbed (XORed) into the state, interleaved with calls of the internal permutation Keccak-f. Once all message blocks are processed, the second phase (called squeezing) starts. In this phase, the first r bits of the state are returned (as hash bits for example). If a desired output is longer than r bits, then the internal permutation is called and then another r bits can be squeezed.

Figure 1 shows how the Keccak state is organised. Terms which denote a given part of the state were introduced by the Keccak designers. For the pseudo-code, it is more convenient to represent the state as the two-dimension array $S[x, y]$, where an element of the array is the 64-bit lane. The default variant of Keccak has the 64-bit lane, but smaller variants (such as the 400-bit state with the 16-bit lane) are also defined. A number of rounds is determined by a size of the state. For the default 1600-bit state, a number of rounds is 24.

In the Keccak permutation all rounds are the same except for the ι step (round-dependent constants XORed into the state). Below we provide a pseudo-code of a single round. Steps in the permutation are denoted by Greek letters.

```
Round(A,RC) {
θ step
C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],              forall x in (0...4)
D[x] = C[x-1] xor rot(C[x+1],1),                                         forall x in (0...4)
A[x,y] = A[x,y] xor D[x],                                       forall (x,y) in (0...4,0...4)

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

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

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

return A   }
```

In the pseudo-code operations on indices are done modulo 5. The state of the permutation is denoted by `A`. There are some intermediate variables such as `B[x,y]`, `C[x]`, `D[x]`. The rotation offsets are `r[x,y]`, whereas `RC` are the round constants. We denote bitwise rotation operation by `rot(W,m)`. It moves a bit at position i into position $i+m$ in the lane `W` ($i+m$ are done modulo the lane size).

θ is a linear operation, a main source of diffusion in the algorithm. ρ is a permutation that mixes bits of a lane, while π permutes the whole lanes. χ can be viewed as a layer of the 5-bit Sboxes. The last step is ι, which XORes the round constant into the first lane. The values of the round constants are generated by a simple linear feedback shift register (LFSR) [6].

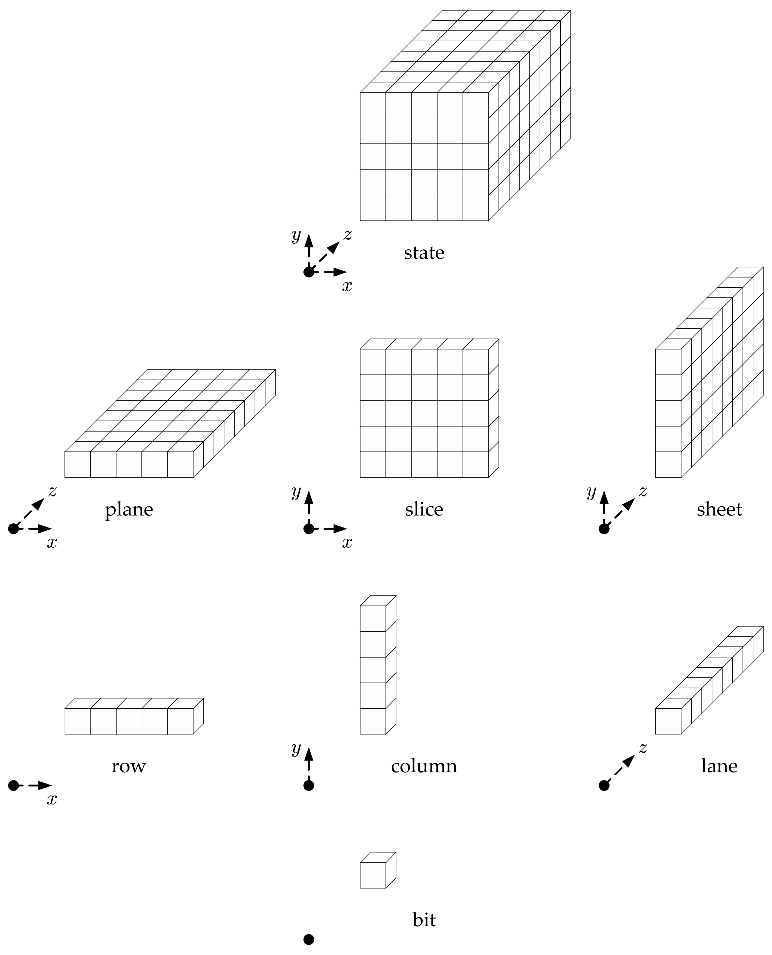

Fig. 1. Pieces of the Keccak state[6]

2.1 Malicious Keccak

For a malicious variant of Keccak, we change the round constants. All other steps and parameters in the algorithm remain the same. Our malicious constants are also generated with a simple shift register. The register is seeded with "SHA3SHA3" (64-bit constant encoded in ASCII). The subsequent round constants are obtained by a rotation of the register by one bit. The constants, given in hexadecimal notation, are as follows.

```
RC[ 0] := 5348413353484133      RC[ 1] := A9A42099A9A42099      RC[ 2] := D4D2104CD4D2104C
RC[ 3] := 6A6908266A690826      RC[ 4] := 3534841335348413      RC[ 5] := 9A9A42099A9A4209
RC[ 6] := CD4D2104CD4D2104      RC[ 7] := 66A6908266A69082      RC[ 8] := 3353484133534841
RC[ 9] := 99A9A42099A9A420      RC[10] := 4CD4D2104CD4D210      RC[11] := 266A6908266A6908
RC[12] := 1335348413353484      RC[13] := 099A9A42099A9A42      RC[14] := 04CD4D2104CD4D21
RC[15] := 8266A6908266A690      RC[16] := 4133534841335348      RC[17] := 2099A9A42099A9A4
RC[18] := 104CD4D2104CD4D2      RC[19] := 08266A6908266A69      RC[20] := 8413353484133534
RC[21] := 42099A9A42099A9A      RC[22] := 2104CD4D2104CD4D      RC[23] := 908266A6908266A6
```

All 24 constants are different, however, a careful reader notices that they are symmetric. This symmetry plays a vital role in attacks on malicious Keccak.

3 Attacks on Malicious Keccak

In the attacks on malicious Keccak we use a variant of differential cryptanalysis namely internal differential cryptanalysis. In typical differential attacks, we consider two different plaintexts and follow an evolution of the differences between them. In case of internal differential attacks only one plaintext is considered and we trace a statistical evolution of the differences between its parts. Such analysis was first proposed by Peyrin [16] in the attack on the Grøstl hash function. In [9], Dinur et al. showed that internal differentials could be also used to produce collisions against round-reduced Keccak.

There is a particular property of Keccak, already noticed by the designers [6], which makes Keccak a promising candidate for internal differential cryptanalysis. That is four out of its five internal mappings (all but ι) are translation invariant in a direction of the z axis. Namely, if one state S is a rotation of another state $S^{'}$ with respect to the z axis (i.e., $S^{'}[x][y][z] = S[x][y][z+i]$, for some value of i), then applying to them any of the θ, ρ, π, χ operations maintains this property. This leads to the following observation. If we divide the state on two halves along z axis, where both halves are the same, θ, ρ, π, χ operations do not destroy the symmetry. What does destroy the symmetry is the ι step, namely XORing the round constants to the state. Here comes a natural question, that is, what happens if we craft the constants such as the above-mentioned property works for every step in the algorithm. This is how we come up with the idea of malicious constants in Keccak. If constants are also symmetrical, as the ones proposed in Section 2.1, the symmetry of the state is kept through all the steps and we can exploit it to devise attacks on the Keccak hash function.

3.1 Collision Attack

Here we present a collision attack on malicious Keccak with the l-bit hash. Applying the standard birthday attack, we can find collisions with $2^{l/2}$ calls. However, for malicious Keccak, a collision can be found with the $2^{l/4}$ effort. In our attack, instead of calling the algorithm with random messages, we use the messages which have symmetric structure. That is, once the message is absorbed into the state, both halves in each lane are the same. This property, as already explained, would be preserved at every step of the algorithm. In particular, hashes would be also symmetric. It means that, in fact, we search for collisions for the $(l/2)$-bit hash. Once found, we are sure that the second parts of lanes in the hash also collide. Hence, a cost of a collision attack is $2^{l/4}$.

There are some technicalities worth discussing. First, the attacker has to consider padding. The last 62 bits of a message has to contain all 1's and a length of the message is equal to the bitrate r minus 2. This way the message is padded with two 1's and the last lane in the bitrate part of the state is filled with all 1's (clearly a symmetric lane).

The attack scales naturally for longer or shorter hashes, for example, the attack on Keccak with the 512-bit hash would cost $2^{512/4} = 2^{128}$. If a hash length is not a multiple of the lane size (64), then the attack does not fully exploit the symmetry property. So, for example, a cost of a collision attack for the 256-bit and 224-bit hash is the same. It is because for the 224-bit hash we don't get an extra 32 bits 'for free' in the last lane of the hash string.

We implemented the attack for the Keccak variant with the 1024-bit bitrate and the 128-bit hash. As expected, after about $2^{128/4}$ calls we found a collision. The colliding (padded) messages and a hash are given below.

Table 1. An example of a collision for malicious Keccak.

m	813e344a813e344a	78d30cf978d30cf9	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	1111111111111111
m'	256a4f71256a4f71	e788dc79e788dc79	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	1111111111111111
hash	a012dcb9a012dcb9	810c79e0810c79e0		

3.2 Preimage Attack

In a similar way to a collision attack, we are able to find preimages for the symmetric hashes. Typically, to convince a third party that we can mount the preimage attack, we show a preimage for some non-random hashes, e.g., all 0's or all bytes identical. Such hashes would be covered by our preimage attack as they are clearly symmetric. In the attack against malicious Keccak with l-bit hash, we search, in fact, the preimage for $l/2$-bit hash, as the second half of the bits are guaranteed to be the same, due to the symmetric property of malicious Keccak. Therefore, the preimage attack complexity is $2^{l/2}$, whereas the exhaustive search costs 2^{l}. As in the collision search, our preimage attack exploits its full potential when a given hash is a multiple of 64.

We found a preimage for malicious Keccak with the 64-bit hash, where all hash bits are 0.

Table 2. A preimage for a given 64-bit hash.

m	7187c4197187c419	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	1111111111111111
hash	0000000000000000			

3.3 Weak Keys in Keccak-MAC

Keccak can also work in the keyed mode, particularly providing the message authentication code (MAC). The hash-based MAC involves a cryptographic hash function in combination with a secret key and a typical solution is HMAC proposed by Bellare et al. [3]. However, in the case of Keccak, a MAC functionality can be done in a more straightforward way then the nested approach of HMAC. A secret key is simply prepended to a message, so for Keccak with the 128-bit key first two lanes of the state are occupied by key bits. Then, the state is processed and a tag is obtained (squeezed).

A secure MAC algorithm should have two main properties. First, it should be infeasible to mount the key-recovery attack, even if the attacker has many valid message-tag pairs. Second, the attacker should not be able to create a forgery, namely, to show a valid message-tag pair (M, T) for a message M that has not been previously authenticated.

For malicious Keccak a forgery attack is possible when a key belongs to the weak keys class. The class consists of symmetric keys, that is the first part of a lane is the same as the second one. To attack the Keccak-MAC with the 128-bit key and 128-bit tag, we provide 2^{64} different pairs of (M, T). Each pair has the same, chosen symmetric tag and different (but also symmetric) message lanes. Because of the symmetric property, one of the 2^{64} chosen pairs will be actually the valid pair, whereas for the secure MAC algorithm we would need 2^{128} tries to provide the valid, previously unseen, message-tag pair. Please note that checking whether a key belongs to the weak keys class requires just a single call of the Keccak-MAC. If a key is in the class, the tag will be symmetric. (There might be false positives but their probability is only 2^{-64} and they can be easily verified with another call of the algorithm, with different message bits.)

3.4 Other Symmetries

The malicious constants we propose are symmetric, that is both 32-bit parts of a lane are the same. We can exploit the symmetry even further and divide a given 64-bit lane on shorter, identical blocks. A size of a block could be 32 (as in our malicious set), 16, 8, 4, 2 — divisors of 64. If a block is smaller, a cost of the attacks is also smaller. This is because we get more bits 'for free'. So, for example, the malicious constants with a block size 16 (four identical blocks in a lane) decreases the cost of our preimage attack to $2^{l/4}$. A drawback of shorter blocks is that we may not guarantee distinct constants for all 24 rounds and they would look less and less random.

3.5 More 'nothing up my sleeve' Constants

To convince a user that chosen constants are indeed 'nothing up my sleeve', we can change the last round constant and set it as, for example, 64 bits taken from π or any other, well known, mathematical constant. Certainly, such a constant is not a symmetric string of bits. But please note that the constant is XORed into to the state as the last step of the permutation, so it only acts as a 'mask'. If there were some subsequent steps in the algorithm, then such the constant would be a beginning of breaking the symmetry in the state. For the collision and forgery attack, this new constant does not affect the attack, we can always 'unmask' obtained hashes (tags) and observe symmetric hashes (tags). In case of the preimage attack, where we want to show a preimage for a chosen (e.g. all 0's) hash, the new, non-symmetric constant spoils the attack or at least it makes the attack less convincing. We could still find preimages, but a chosen hash would have been 'affected' by the π constant. For example, we could find a preimage for a hash with its first 64-bits equal to π concatenated with the all-zero vector.

4 Differential Fault Analysis

In this section, we show how the idea of malicious constants in Keccak can be used to mount differential fault analysis and recover the secret key. Differential fault analysis (DAF) is a type of side-channel attack, where the attacker is able to induce faults (due to high temperature, unsupported supply voltage or current, or other factors) into the state of an algorithm [8]. There have been many models of DAF, motivated by different hardware implementations and setups of a cryptographic algorithm. Typically, we assume that an induced fault changes a value of a bit or a byte to a random value. Recently, it has been shown that for many DRAM memories, the attacker can flip particular bits without accessing to them [13]. Since DRAM process technology reaches smaller and smaller dimensions, it becomes more difficult to prevent DRAM cells from electrically interacting with each other.

We attack Keccak (SHA-3) working in the authenticated encryption mode. The idea of using the sponge function to provide both integrity and confidentiality was first given by Keccak designers in [7]. Figure 2 shows an example scheme.

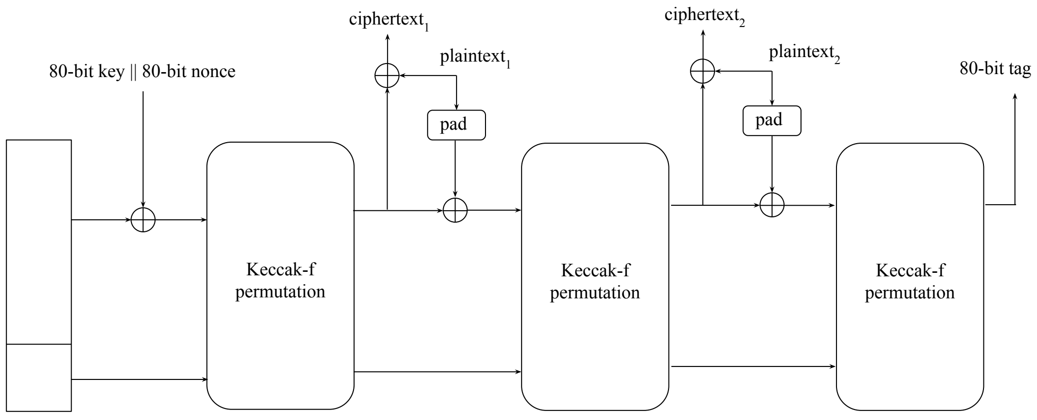

Fig. 2. Keccak working in AE mode processing two plaintext blocks.

In our attack, we focus on the Keccak variant with 80-bit key and 400-bit state. A size of a lane is 16 bits. We respect the nonce requirement, that is every call to algorithm (with induced fault or not) uses a different nonce. (Please note that differential fault analysis on SHA-3 reported in [2] does not cover a scenario where a nonce is respected.)

The aim of our attack is to recover the secret key. We assume that the attacker is able to flip some selected bits of the initial state and flip some constant bits. We are aware that for many hardware implementations this model is unrealistic but for some scenarios such as the above-mentioned modern DRAM memories the attack could be implemented. The idea behind the attack is to force the state into the desired symmetry, as described in previous sections. Once the state is symmetric, the squeezed output (ciphertext blocks) will also exhibit the symmetry, giving a hint that the induced faults has led to the 'correct' modifications of key bits.

First, bits in the constants are flipped such as the constants become symmetric (For the 12-round variant it requires 27 single-bit flips.) We could assume that this is done only once as the constants are stored in memory as a look-up table. The 80-bit secret key is placed in the first 5 lanes of the state and the attacker manipulates the values of the second halves of these lanes. If she hits the right combination, the corresponding ciphertext blocks will be symmetric. As the attacker manipulates 40 bits (a half of all key bits), she needs 2^{40} tries to force the key to be symmetric. To minimize a number of flips (fault inductions), one could check subsequent combinations in the Grey code manner. Note that the 'online' phase of the attack gives us only information, which bits (among those 40 bits) need to be flipped to have a symmetric key. Therefore, later (offline) we have to check 2^{40} keys to get the right one. So, to recover 80-bit secret key we need, on average, 2^{39} faulty encryptions plus 2^{40} offline calls to the algorithm.

Practicality of this attack heavily depends on the model, hardware implementation, latency between subsequent fault inductions, etc. Here, rather than focusing on a particular scenario and hardware setup, we wanted to convey the idea how malicious Keccak is (potentially) helpful in differential fault analysis.

5 Constants Crafted for Differential Characteristic

In the previous section, we proposed the malicious constants exploiting the symmetric nature of the Keccak internal mappings. It leads to some efficient attacks, however, one may argue that the symmetric constants would be quickly noticed and the malicious attacks behind it would be discovered. Here, we take a different approach, namely investigating internal differential characteristics and crafting the constants in such a way that they would increase the probability of a given characteristic. The same approach was taken in the work on malicious SHA-1 [1].

5.1 Differential Characteristic Search

One of the best strategy for building a good differential path for Keccak is to treat each slice of the state separately [10]. The idea behind this approach is that when all columns in a slice have an even parity of bit differences (column parity kernel), diffusion in the θ step is limited. Consequently, fewer Sboxes are active in the subsequent χ step and probability of a differential is higher. A set of malicious constants we craft here should help a given slice to stay in the column parity kernel.

As we deal with internal differentials, we could start with a zero-difference state. In the first round we keep the original Keccak constant, namely 0x0000000000000001. A transition of a given slice through the non-linear χ is done as follows.

1. Find all possible output differences by looking into the difference distribution table of the Keccak Sbox.
2. For each solution found in Step 1, create another solution by changing the bit [0,0] in a slice.
3. Choose the output difference, which gives a slice with a maximum number of columns with an even parity. (For draws, use a number of active bits in a slice as an additional criterion.)

If a chosen output difference belongs to a set created in Step 2, then a bit in a round constant corresponding to a particular slice is set to 1. This is how we build our characteristic and craft the constants along the way. Figure 3 shows an evolution of a differential path and the chosen constants for the first 5 rounds.

Our investigation shows that even if an attacker is able to control the constants, it is hard to build a long differential path with a high probability. It is because the state is large (1600 bits) and the 64-bit constant is just 1/25 of the whole state. So our 'corrections' have quite limited impact. On the contrary, constants in SHA-1 are (relatively) much bigger (1/5 of the state), thus it is much easier to correct the differences [1]. This observation supports the claim that

Table 3. An evolution of internal differential path for malicious Keccak.

Initial state of internal differences

00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000

$\downarrow$

1st round (0 active Sboxes, $p = 2^{0}$)
Round constant: 0000000000000001

$\downarrow$

10000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000	00000000

$\downarrow$

2nd round (11 active Sboxes, $p = 2^{-22}$)
Round constant: 0000000088000001

$\downarrow$

01000008	00000100	00000000	00000000	00002000
00000000	00800000	00000000	00008000	00000000
00000008	00000000	00000000	00000800	00000000
01000000	00000000	00000400	00000000	00000000
00000000	00000000	00000010	00000000	00000004

$\downarrow$

3rd round (78 active Sboxes, $p = 2^{-163}$)
Round constant: 00000000c8242a21

$\downarrow$

20028540	00200008	18800000	a0100080	02001000
20000141	80000008	08081800	00500000	00040aa0
00008500	020d1008	010a080e	00880000	30029010
00000411	860c0081	10820004	80000000	12040084
00020510	042110a9	00011000	20600080	20028230

$\downarrow$

4th round (117 active Sboxes, $p = 2^{-290}$)
Round constant: 00000000c8eeba23

$\downarrow$

0016110c	00009108	c3014120	000c0800	0ac04082
00110008	8a000ac0	00801002	600c003c	8a8aa801
00000084	c970a012	40804002	a9008410	25060004
00440100	03401592	e7202320	800800ad	88216004
00431080	40302e48	64312040	49088c81	296588a3

$\downarrow$

5th round (127 active Sboxes, $p = 2^{-336}$)
Round constant: 00ff0b00ed00f0bb

$\downarrow$

5b617c91	009a0000	08004821	85401020	122084c1
00210105	001aa038	00020080	c0841872	03800009
04004014	03810080	00ad0803	21400408	003042c0
50401800	c1558088	082500ea	4600c74c	000a081e
0f002580	c2542030	008a4048	2284eb16	119ace16

the Keccak permutation is strong against differential cryptanalysis and its diffusion (provided mainly by the θ step) cannot be easily limited.

Recently, it has been shown that using a variant of differential cryptanalysis (the boomerang attack) helps to build the 8-round differential path with a very low complexity [12]. However, that analysis was only given for the Keccak permutation and the attack cannot be converted to the hash function mode. The reason is that when we consider the Keccak hash function, the capacity part of the state is out of control of the attacker and differences cannot be set there.

6 Conclusion

In this paper, we investigated a malicious version of the Keccak hash function. We propose a new set of constants, which exploits a symmetric nature of the Keccak permutation and allows the collision, preimage and forgery attacks, substantially faster than generic ones. Our malicious set of constants has some space for bringing more ‘nothing up my sleeve’ numbers. One can take a well known mathematical constant and set it in the 24th round (or even 23rd round). However, it limits an efficiency of the attacks since the symmetry of the state would be disturbed. Our results do not threaten the original SHA-3 standard, however, they clearly show a user should be very careful with the ‘enhanced’ or ‘personalised’ variants of the algorithm, advertised in a commercial cryptography package.

Additionally, we showed how the idea of malicious Keccak could be used in differential fault analysis against real Keccak (SHA-3) working in the keyed mode.

Acknowledgement

Project was financed by Polish National Science Centre, project DEC-2013/09/D/ST6/03918.

References

1. Albertini, A., Aumasson, J., Eichlseder, M., Mendel, F., Schläffer, M.: Malicious hashing: Eve's variant of SHA-1. In: Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers. pp. 1–19 (2014)
2. Bagheri, N., Ghaedi, N., Sanadhya, S.K.: Progress in Cryptology – INDOCRYPT 2015: 16th International Conference on Cryptology in India, Bangalore, India, December 6-9, 2015, Proceedings, chap. Differential Fault Analysis of SHA-3, pp. 253–269. Springer International Publishing, Cham (2015)
3. Bellare, M., Canetti, R., Krawczyk, H.: Message Authentication Using Hash Functions: the HMAC Construction. CryptoBytes 2(1), 12–15 (1996)
4. Bernstein, D.J., Lange, T., Niederhagen, R.: Dual EC: A Standardized Back Door. Cryptology ePrint Archive, Report 2015/767 (2015), `http://eprint.iacr.org/`
5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Cryptographic Sponges, `http://sponge.noekeon.org/CSF-0.1.pdf`
6. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak Sponge Function Family Main Document, `http://keccak.noekeon.org/Keccak-main-2.1.pdf`
7. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: Duplexing the Sponge: Single-pass Authenticated Encryption and Other Applications. Cryptology ePrint Archive, Report 2011/499 (2011), `http://eprint.iacr.org/`
8. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings. pp. 513–525 (1997)
9. Dinur, I., Dunkelman, O., Shamir, A.: Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials. In: Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers. pp. 219–240 (2013)
10. Duc, A., Guo, J., Peyrin, T., Wei, L.: Unaligned Rebound Attack - Application to Keccak. Cryptology ePrint Archive, Report 2011/420 (2011)
11. Federal Agency on Technical Regulation and Metrology (GOST): Gost r 34.11-2012: Streebog hash function. www.streebog.net (2012)
12. Jean, J., Nikolic, I.: Internal Differential Boomerangs: Practical Analysis of the Round-Reduced Keccak-f Permutation. In: Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers. pp. 537–556 (2015)
13. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. pp. 361–372. ISCA '14 (2014)
14. Koblitz, N., Menezes, A.: A riddle wrapped in an enigma. Cryptology ePrint Archive, Report 2015/1018 (2015), `http://eprint.iacr.org/`
15. Morawiecki, P., Pieprzyk, J., Srebrny, M.: Rotational cryptanalysis of round-reduced Keccak. In: Fast Software Encryption. LNCS, Springer (2013)
16. Peyrin, T.: Improved Differential Attacks for ECHO and Grøstl. In: Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings. pp. 370–392 (2010)
17. Rijmen, V., Preneel, B.: A Family of Trapdoor Ciphers. In: Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings. pp. 139–148 (1997)
18. Young, A.L., Yung, M.: Malicious cryptography - exposing cryptovirology. Wiley (2004)