Side-Channel Analysis of MAC-Keccak Hardware Implementations

Pei Luo[1], Yunsi Fei[1], Xin Fang[1], A. Adam Ding[2], David R. Kaeli[1], and Miriam Leeser[1]

[1] Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115
silenceluo@coe.neu.edu, yfei@ece.neu.edu, fang.xi@husky.neu.edu, kaeli@ece.neu.edu, mel@coe.neu.edu

[2] Department of Mathematics, Northeastern University, Boston, MA 02115
a.ding@neu.edu

Abstract. As Keccak has been selected as the new SHA-3 standard, Message Authentication Code (MAC) (MAC-Keccak) using a secret key will be widely used for integrity checking and authenticity assurance. Recent works have shown the feasibility of side-channel attacks against software implementations of MAC-Keccak to retrieve the key, with the security assessment of hardware implementations remaining an open problem. In this paper, we present a comprehensive and practical side-channel analysis of a hardware implementation of MAC-Keccak on FPGA. Different from previous works, we propose a new attack method targeting the first round output of MAC-Keccak rather than the linear operation θ only. The results on sampled power traces show that the unprotected hardware implementation of MAC-Keccak is vulnerable to side-channel attacks, and attacking the nonlinear operation of MAC-Keccak is very effective. We further discuss countermeasures against side-channel analysis on hardware MAC-Keccak. Finally, we discuss the impact of the key length on side-channel analysis and compare the attack complexity between MAC-Keccak and other cryptographic algorithms.

Keywords: AES, differential fault analysis, side-channel attacks

1 introduction

Keccak was selected as the winner of the NIST hash function competition in October, 2012 to be the new hash standard SHA-3 [1]. Different from previous hash functions, Keccak uses the *Sponge construction* in which message blocks are absorbed and permuted iteratively [2,3,1]. A message authentication code (MAC) is a short piece of information generated by hash functions on the message and a secret key. MAC is important in crypto systems because it facilitates integrity checking and proof of origin for the message. The Sponge construction allows Keccak to securely generate a MAC by hashing the concatenation of the key and the message ($P = K||M$) in a cryptographic mode, called MAC-Keccak [2,3].

With the new algorithms of Keccak and MAC-Keccak, their security issues have attracted a lot of attention. As the new hash standard, Keccak will be widely used in various cryptographic systems, and thus its side-channel security becomes a critical issue. There exist several related work to assess the side-channel security of MAC-Keccak. In [4], six SHA-3 candidates (including Keccak) were analyzed for side-channel leakage for the first time. The work analyzes the possibility of attacking MAC-Keccak at the θ step in the first round and proposes the basic attack steps. However, only general side-channel attack feasibility for all six candidates is discussed, and no detailed leakage models or attack methods for Keccak are given.

Some works specifically on Keccak are published after that. In [5] and [6], the side-channel vulnerability of MAC-Keccak software implementations is analyzed and it is found that the side-channel resistance of MAC-Keccak depends on the length of the key. The works demonstrate a practical side-channel analysis attack on MAC-Keccak implemented on a 32-bit Microblaze processor, which also attacks the θ step. In a previous paper [7], we find side-channel leakages in hardware implementations based on the properties of Keccak and the implementation details. We find that attackers can use the same models and attack methods as in software implementations to conquer the hardware implementations.

These previous works either focus on software implementations [6,5] or use the models for software implementations to attack hardware systems [7], while no side-channel methods specifically for hardware implementations of Keccak have been proposed. Compared to MAC-Keccak software implementations, hardware implementations of MAC-Keccak have different operation granularity and storage methodology, thus some previous models for software systems are not suitable for hardware implementations, and there should be some different attacking methods for hardware implementations. Since hardware implementations of Keccak will surely be used extensively in performance-sensitive crypto systems, their side-channel leakage must be thoroughly assessed and more general side-channel attacking methods for hardware implementations should be proposed. Furthermore, a good understanding of the side-channel leakage will also facilitate secure hardware MAC-Keccak design and implementation with protections against side-channel attacks added.

In this paper we analyze the algorithm of Keccak and design a new attack method on the first round output of MAC-Keccak. We propose new side-channel leakage models specifically for hardware MAC-Keccak implementations and launch practical power analysis attacks. Results show that our method can recover the key bits efficiently. We further discuss the factors that affect side-channel leakage of MAC-Keccak and suggest countermeasures to improve the system security.

To our knowledge, this is the first attempt to launch side-channel attacks on the first round output of MAC-Keccak. Our work should provide a deeper and more precise understanding of the side-channel leakage of Keccak. We expect our work to make a significant contribution to security evaluation of SHA-3 and help to improve its resilience to side-channel attacks.

The rest of this paper is organized as follows. In Section 2, we introduce background of Keccak. In Section 3, we present our power leakage model and attack method on the first round output R_1, followed by the attack results. In Section 4, we discuss the attack complexity and protection methods of MAC-Keccak. In Section 5, we conclude the paper.

2 Details of Keccak

2.1 Keccak Hash Function and MAC-Keccak

Keccak is a hash function family based on the Sponge construction, as shown in Figure 1 [2,3]. Keccak has two phases: 1) absorbing and 2) squeezing. In the absorbing phase, the message is broken into blocks (each block size is r bits, where r is the bit rate), which are absorbed iteratively by the permutation function f. Each f function works on a state at a fixed length $b = r + c$, where r is the bit rate while c is capacity. In the squeezing phase, outputs are squeezed also by f functions and the length of the output is configurable (a multiple of r bits).

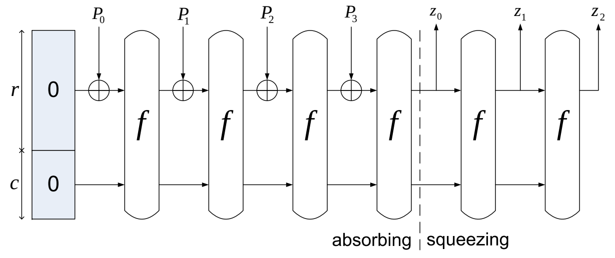

Fig. 1: The sponge construction

For Keccak, MAC-Keccak is recommended by the Keccak designers [2,3]:

$$\mathsf{MAC(M,K)} = \mathsf{H}(K||M). \tag{1}$$

The default Keccak mode is Keccak-1600, with $r = 1024$ and $c = 576$ [2,3]. All of the 1600-bit states are organized in a 3-D array, as shown in Figure 2. Each bit is addressed with three coordinates, written as $S(x, y, z),\ x, y \in \{0, 1, ..., 4\},\ \ z \in \{0, 1, ..., 63\}$. 2-D entities, *plane*, *sheet* and *slice*, and 1-D entities, *lane*, *column* and *row*, are also defined in Keccak and shown in Figure 2. A plane P_Y contains all state bits $S(x, y, z)$ for which $y = Y$; similarly, a slice $SL_Z = \{S(x, y, z),\ z = Z\}$; a sheet $SH_X = \{S(x, y, z),\ x = X\}$. A lane $L_{X,Y}$ contains all state bits $S(x, y, z)$ for which $x = X$ and $y = Y$; similarly, a column $C_{X,Z} = \{S(x, y, z),\ x = X,\ z = Z\}$; a row $Row_{Y,Z} = \{S(x, y, z),\ y = Y,\ z = Z\}$.

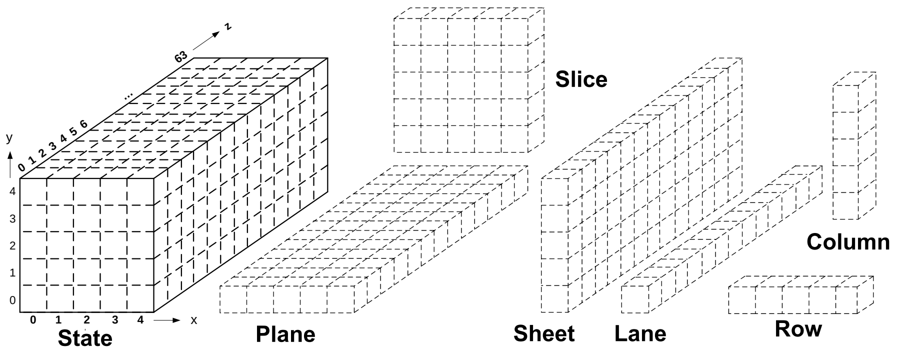

Fig. 2: Terminology used in Keccak

The f permutation function of Keccak-1600 consists of 24 rounds of operations, where each round has five sequential steps:

$$R_{i+1} = \iota \circ \chi \circ \pi \circ \rho \circ \theta(R_i),\ i \in \{0, 1, \cdots, 23\} \tag{2}$$

in which R_0 is the initial input. Details of each step are described below:

– θ is a linear operation which involves 11 input bits and outputs a single bit. Each output state bit is the XOR between the input state bit and two intermediate bits produced by its two neighbor columns. The operation is given as follows:

$$\begin{aligned} S'(x, y, z) = S(x, y, z) &\oplus (\oplus_{i=0}^{4} S(x-1, i, z)) \\ &\oplus (\oplus_{i=0}^{4} S(x+1, i, z-1)). \end{aligned} \tag{3}$$

The two intermediate bits are the parity of the two columns, $\oplus_{i=0}^{4} S(x-1, i, z)$ and $\oplus_{i=0}^{4} S(x+1, i, z-1)$, respectively.

– ρ is a permutation over the bits of the state along the z-axis (in lanes).

– π is a permutation over the bits of the state within slices, shown in Figure 3. Only the center bit $(x = 0,\ y = 0)$ of the slice does not move. All other bits are permuted to other positions depending on their original coordinates. We also notice that five bits of each row will be moved to different rows but the same column. For each output row of π, one and only one bit is from the bottom row $(y = 0)$ of the input.

– χ is a non-linear step that contains mixed binary operations. Every bit of the output state is the result of an XOR between the corresponding input state bit and its two neighboring bits along the x-axis (in a row):

$$S'(x, y, z) = S(x, y, z) \oplus (\overline{S(x+1, y, z)} \cdot S(x+2, y, z)). \tag{4}$$

– ι is a binary XOR with a round constant.

Further details of Keccak and Sponge construction can be found in [2,3,1].

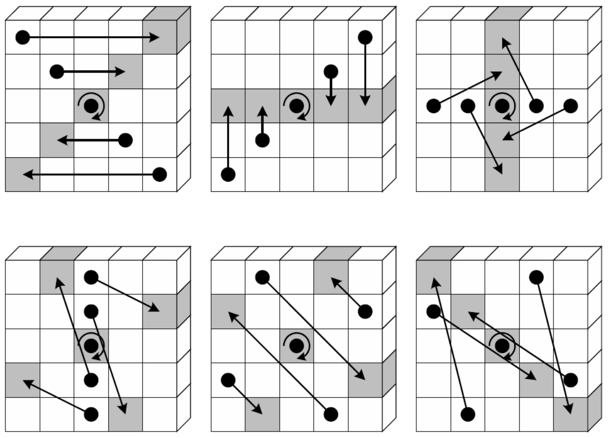

Fig. 3: π applied to a slice. Note that $x = y = 0$ is depicted at the center of the slice.

For the official VHDL implementation Version 3.1 provided online by *keccak.noekeon.org* [8] and other widely used hardware implementation [9], each round takes one clock cycle and the five steps of a round (θ, ρ, π, χ and ι) are all implemented in combinational circuits. In this paper we focus on the final output of the first round for power analysis.

2.2 MAC-Keccak Parameters Used in This Paper

MAC-Keccak allows the use of a variable-length key, which is concatenated with the message and then broken down into message blocks (each at the bit rate, and the last one is padded according to a certain rule [2]). We start with a key length of 320 bits (i.e., the key fills the bottom plane in the input state, and the message length is $1024 - 320 = 704$ bits). We will discuss the effects of key length on attacks in Section 4.3.

For these settings, the key bits that fill the first plane (denoted as the K plane) of the input state are unknown and the other four planes (denoted as the M planes) that contain message bits and padding bits are known to the attackers. For the θ operation, each bit of θ_{out} in the M planes involves eight bits of M and two bits of K.

In this paper, we use the same traces as [7] provided online [10]. These traces are sampled from an implementation of [8] on an SASEBO-GII board. More details of the implementation can be found in [7]. We note here that we use both mutual information [11] and CPA for analysis, their results are very similar and we only present and discuss the CPA results in this paper.

3 Side-channel power analysis on the first round output R_1

Previous papers [7,6,5,4] about side-channel attacks of Keccak all focus on the θ operation of the first round because it involves key bits directly and it's the first step of Keccak. It has also been discussed in [7] that leakages of θ may still exist in hardware systems because of the mathematical properties of Keccak. Meanwhile, the leakages of θ mentioned in [7] are mainly caused by the leakages of state register activity and it has also been pointed out that the correlation between the first round output Hamming weight and the power consumption is very strong. Thus in this section, we'll try to find attacking methods utilizing the first round result R_1 directly.

3.1 Side-Channel Leakages of R_1

We denote the first round output of Keccak as R_1, and denote the content of state register Reg before the end of first round as I. Then the Hamming distance of Reg at the end of first round is $HD(I, R_1)$. For unprotected Keccak, the initial state I of state register Reg can be either 0, R_0 or the result of last Keccak operation. For the implementation we referred for design [12], the state register Reg is initialized to 0 and thus the Hamming distance is actually $HW(R_0)$. In this section, we focus on this implementation, and we will discuss other situations in Section 4. We show the correlation result between $HW(R_0)$ and power in Figure 4.

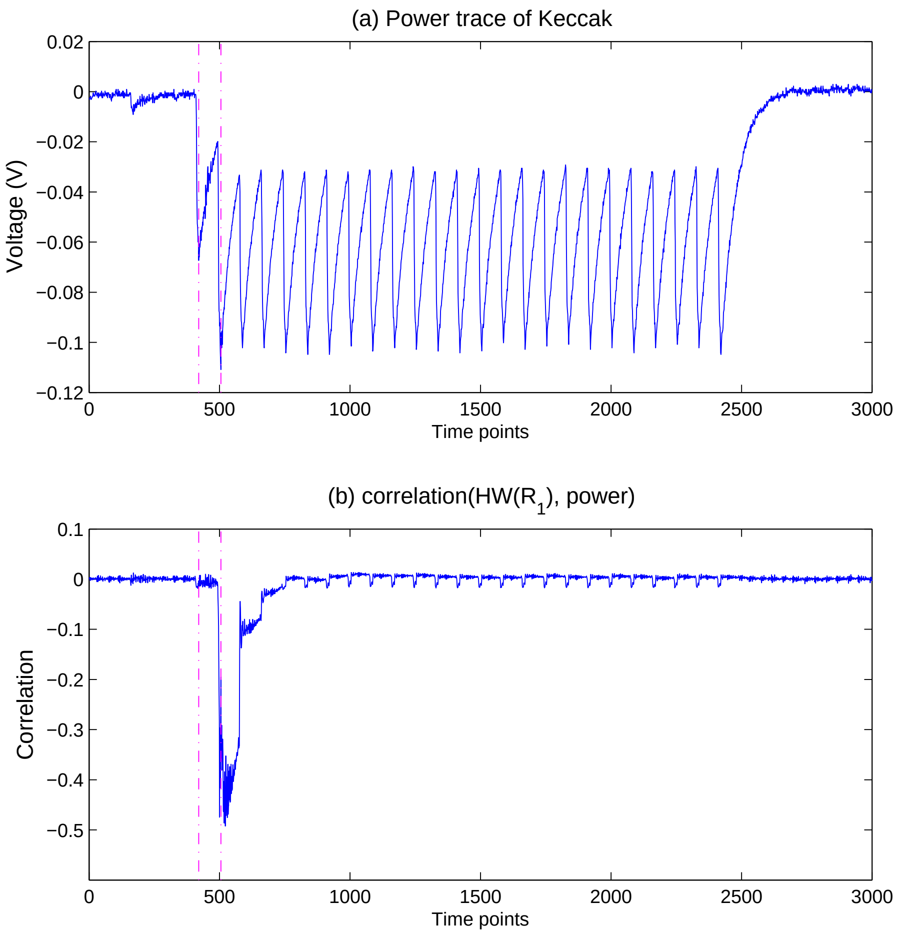

Fig. 4: One Keccak power trace and the correlation between $HW(R_1)$ and power.

Figure 4(a) shows one power trace and Figure 4(b) is the correlation result between $HW(R_1)$ and power consumption. It shows that the correlation between $HW(R_1)$ and power is very strong and this leakage definitely can be used by an attacker to extract secret information. We will use Hamming distance and Hamming weight as selection functions in this paper for analysis.

First of all, we focus on DPA and use one single bit $R_1(x_1, y_1, z_1)$ for analysis, where $x_1, y_1 \in \{0, 1, \cdots, 4\}$ and $z_1 \in \{0, 1, \cdots, 63\}$. The corresponding selection function for this bit is:

$$HD(I(x_1, y_1, z_1), R_1(x_1, y_1, z_1)) = HW(R_1(x_1, y_1, z_1)).$$

As discussed in Section 2.1, $R_1(x_1, y_1, z_1) = \iota_0(x_1, y_1, z_1) \oplus \chi_{out}(x_1, y_1, z_1)$, where ι_0 stands for the constant number of ι operation in the first round, which is public to users. Thus attacking R_1 is also attacking χ operation result χ_{out}. Meanwhile, every bit of $\chi_{out}(x_1, y_1, z_1)$ is decided by three bits of

$\pi_{out}(x_1, y_1, z_1)$:

$$\chi_{out}(x_1, y_1, z_1) = \pi_{out}(x_1, y_1, z_1) \oplus (\overline{\pi_{out}(x_1+1, y_1, z_1)} \\ \cdot \pi_{out}(x_1+2, y_1, z_1)). \tag{5}$$

For Keccak, π and ρ operations only change the position of bits while keep their value unchanged. Thus each bit of $\pi_{out}(x_1, y_1, z_1)$ can be mapped to one bit of θ_{out}. If we assume that $\pi_{out}(x_1, y_1, z_1)$, $\pi_{out}(x_1 + 1, y_1, z_1)$ and $\pi_{out}(x_1 + 2, y_1, z_1)$ are mapped to $\theta_{out}(x_0^0, y_0^0, z_0^0)$, $\theta_{out}(x_0^1, y_0^1, z_0^1)$ and $\theta_{out}(x_0^2, y_0^2, z_0^2)$ respectively, then we have:

$$\chi_{out}(x_1, y_1, z_1) = \theta_{out}(x_0^0, y_0^0, z_0^0) \oplus (\overline{\theta_{out}(x_0^1, y_0^1, z_0^1)} \\ \cdot \theta_{out}(x_0^2, y_0^2, z_0^2)). \tag{6}$$

Thus each bit of R_1 can be treated as the operation result of three bits of θ_{out}. As shown in Section 2.1 and [7], each θ_{out} bit involves two or three key bits. It has also been shown in [7] that each bit of θ_{out} can leak XOR relationship of these two or three bits. Thus by attacking one bit of R_1, attackers should be able to extract three XORs and each XOR includes two or three key bits. For example, for $\theta_{out}(x_0^0, y_0^0, z_0^0)$, the attacker can extract:

$$\begin{cases} \text{If } y_0^0 = 0: \\ \quad S(x_0^0 - 1, 0, z_0^0) \oplus S(x_0^0, 0, z_0^0) \oplus S(x_0^0 + 1, 0, z_0^0 - 1) \\ \text{If } y_0^0 = 1: \\ \quad S(x_0^0 - 1, 0, z_0^0) \oplus S(x_0^0 + 1, 0, z_0^0 - 1) \end{cases}.$$

For $\pi_{out}(x_1, y_1, z_1)$, the corresponding coordinates of θ_{out}, (x_0^0, y_0^0, z_0^0), can be obtained by running the reverse of π and ρ, which we denote as π^{-1} and ρ^{-1} here.

For DPA attacks on R_1 which utilizes leakage of one single bit, the signal to noise ratio (SNR) should be very low. Meanwhile, the redundancy of attacking bit by bit is very high, because there are only 320 different 2-bit XORs and 320 different 3-bit XORs while there are $1,600$ bits for R_1 and each bit of R_1 will involve three XORs. In the following part, we will present a more efficient and more practical method for attacking R_1.

3.2 Practical Attacks on R_1

It is inefficient to attack R_1 bit by bit because of the high redundancy, and it will require a large number of traces because of the low SNR. Meanwhile, we notice that five bits in each row of π_{out} operate on each other to generate χ_{out}, which means that five bits in each row of χ_{out} actually involve only five bits of θ_{out}, instead of 5×3 bits. Thus if we attack on five bits in one row together, the redundancy will be much lower than attacking bit by bit. What's more, the SNR of leakage for one row should be definitely higher than the leakage of one bit. We run correlation power analysis based on $500,000$ traces and show the correlation results of one row ($R_1([0:4], 4, 0)$) and one bit ($R_1(0, 4, 0)$) in Figure 5.

For one row, the maximum correlation between $HW(R_1([0:4], 4, 0))$ and power consumption is about 0.042; while for one single bit it is about 0.022. It's obvious that the leakage of one row is much stronger than the leakage of one bit. Meanwhile, there are 320 rows in Keccak and by attacking row by row, attackers only need to attack 320 times instead of run the attacks 1600 times in attacking bit by bit. For side-channel attacks targeting one row each time, the selection function is:

$$HD(I(X, y, z), R_1(X, y, z)) = HW(R_1(X, y, z)), \\ X = [0:4], y \in \{0, 1, 2, 3, 4\}, z \in \{0, 1, \cdots 63\}. \tag{7}$$

As shown in Figure 3 and Section 2.1, five bits of each row of ρ_{out} are all permuted to all different planes. What's more, each row of π_{out} has one and only one bit from bottom plane of ρ_{out}. Thus there is one and only one bit in each row of π_{out} coming from the bottom plane of θ_{out}, and this bit involves

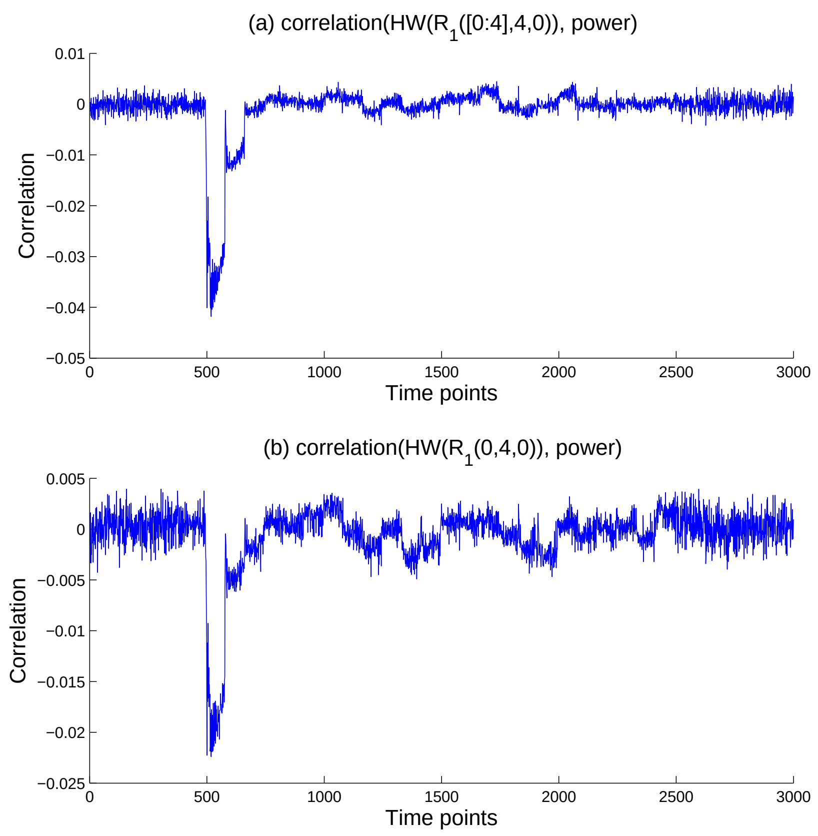

Fig. 5: The correlation between five bits vs. one bit of R_1 and power

three key bits instead of two. Thus, by attacking one row of R_1, attackers can extract one 3-bit XOR and four 2-bit XORs.

The attacking method discussed in Section 3.1 and 3.2 can be used to attack the first round output of Keccak, R_1. We run CPA on the traces using our method and results show that the proposed method can efficiently recover the secret key information. We give an example to demonstrate the attacking method and the results.

Example 1. For the row $R_1([0:4],4,0)$, based on π^{-1} and ρ^{-1}, we can get mapping relationship as following:

$$\begin{cases} \pi_{out}(0,4,0) \leftrightarrow \theta_{out}(2,0,2) \\ \pi_{out}(1,4,0) \leftrightarrow \theta_{out}(3,1,9) \\ \pi_{out}(2,4,0) \leftrightarrow \theta_{out}(4,2,25) \\ \pi_{out}(3,4,0) \leftrightarrow \theta_{out}(0,3,23) \\ \pi_{out}(4,4,0) \leftrightarrow \theta_{out}(1,4,62) \end{cases}.$$

As $\theta_{out}(2,0,2)$ is on the bottom plane, it involves $P(2,0,2)$, $P(1,0,2)$ and $P(3,0,1)$, three key bits. The other four θ_{out} bits each involves two key bits. Using this attack model, for each row of R_1 we can recover one XOR of three key bits and four XORs of two key bits. In this example, we can recover:

$$\begin{cases} kg_1 = P(2,0,2) \oplus P(1,0,2) \oplus \ P(3,0,1) \\ kg_2 = P(2,0,9) \oplus P(4,0,8) \\ kg_3 = P(3,0,25) \oplus P(0,0,24) \\ kg_4 = P(4,0,23) \oplus P(1,0,22) \\ kg_5 = P(1,0,62) \oplus P(2,0,61) \end{cases}.$$

Using the Hamming weight power model $HW(R_1([0:4],4,0))$ to run CPA, the correlation result between $HW(R_1([0:4],4,0))$ and power for different key guesses is shown in Figure 6(a), where the Y-axis is the correlation value and X-axis is the enumeration (key guess) of $[kg_5\ kg_4\ kg_3\ kg_2\ kg_1]$, with kg_5 as the most significant bit.

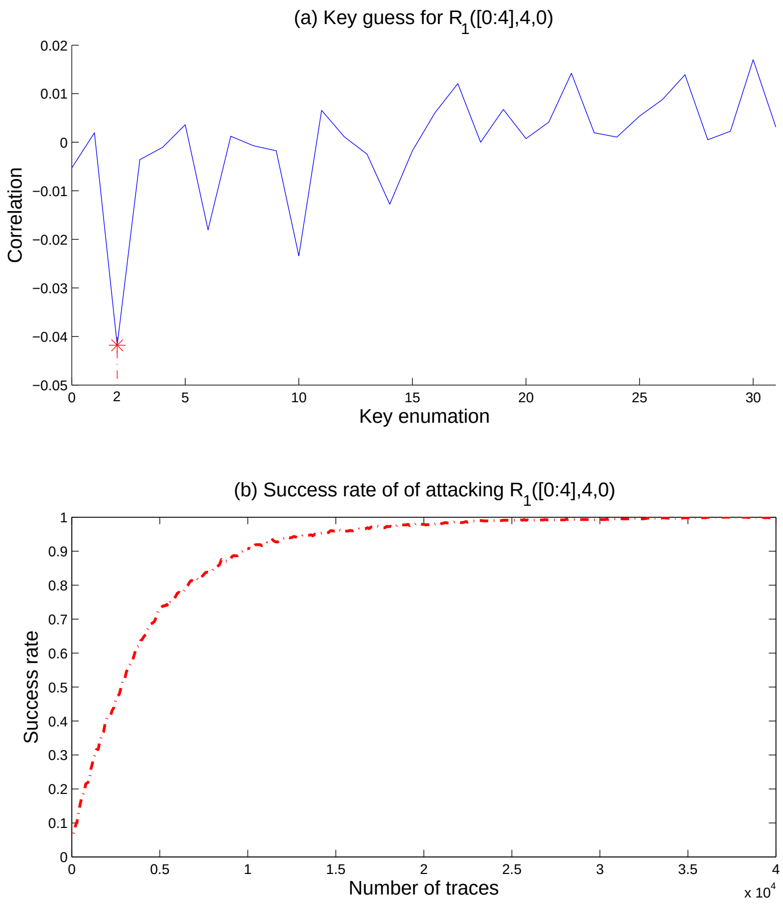

Fig. 6: CPA trace on R_1

For Figure 6(a), the highest correlation value (negative) is -0.042, and is indeed at the correct guess. We can see that our attack model can effectively distinguish a group of five correct results of key bits' operations from each row. The correlation result shown in Figure 6(a) is quite different from the previous block ciphers (*e.g.*, AES, DES) where the correct key would give a much higher correlation than other incorrect keys. This is because MAC-Keccak does not have modules similar to S-box which has very high nonlinearity, therefore the difference between key guesses is not as large as in block ciphers.

Figure 6(b) shows the attack success rate of the row $R_1([0:4],4,0)$ based on our method. It shows that the attacker needs about $35,000$ traces to recover the corresponding key bits relationships, which include one XOR result of three key bits and four XOR results of two key bits, with a success rate of 100%.

In Example 1, we use the row $R_1([0:4],4,0)$ as an example to demonstrate the CPA result and success rate of attacking the hardware implementation of MAC-Keccak based on our attack method. We note here that CPA on other rows has similar results. The above analysis and Example 1 shows that by attacking one row of R_1 attackers can obtain relationship between key bits, but not key bits directly. Multiple rows must be utilized together to obtain key bits. In our attack method, the piecemeal information obtained from attacking each row has to be stitched together to retrieve the key bits.

Example 2. If we attack on row $R_1([0:4],2,8)$ using the same method as Example 1, we can recover:

$$\begin{cases} kg'_1 = P(1,0,7) \oplus P(0,0,7) \ \oplus \ P(2,0,6) \\ kg'_2 = P(1,0,2) \oplus P(3,0,1) \\ kg'_3 = P(2,0,41) \oplus P(4,0,40) \\ kg'_4 = P(3,0,0) \oplus P(0,0,63) \\ kg'_5 = P(4,0,54) \oplus P(1,0,53) \end{cases}.$$

Combining with the result of attacking $R_1([0:4],4,0)$ in Example 1, we can see that kg'_2 $(P(1,0,2) \oplus P(3,0,1))$ is part of kg_1 $(P(2,0,2) \oplus P(1,0,2) \oplus P(3,0,1))$, thus they can be combined together to recover $P(2,0,2)$.

The output state R_1 consists of 320 rows, and these 320 rows can be attacked in parallel and their results have to be utilized together to recover the key bits. For 320 rows, we can obtain 320 3-bit XORs and 1280 2-bit XORs in total. For these 1280 2-bit XORs, the redundancy is very high and there are only 320 distinct 2-bit XORs with each appearing four times. We can extract all these XORs in two groups as follows:

$$\begin{cases} KG_1 = \{S(x-1,0,z) \oplus S(x,0,z) \oplus S(x+1,0,z-1)\} \\ KG_2 = \{S(x-1,0,z) \oplus S(x+1,0,z-1)\} \end{cases}$$

for $x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\}$.

Because each of the 320 3-bit XORs from group KG_1 contains one extra key bit than one 2-bit XOR from group KG_2, we can combine them to recover all of the 320 key bits immediately. Note that we can also insert the recovered key bits back into 2-bit XORs to recover other key bits. For the case where the key length is greater than 320 bits, we will discuss the extension of our approach in Section 4.3.

According to the above discussions and examples, we conclude the attack method using R_1 as following:

1. Attack one row (Example 1):
   - For one row $R_1(X, y_0, z_0)$, locate the corresponding θ_{out} bit for each bit of $\pi_{out}(X, y_0, z_0)$ according to π^{-1} and ρ^{-1}.
   - For each bit of θ_{out}, make assumption of the XOR of the corresponding key bits and combine the assumption with known bits (P and M) to calculate this θ_{out} (π_{out}) bit.
   - Combine all the related θ_{out} (π_{out}) bits to generate $R_1(X, y_0, z_0)$, and run CPA for all the possible XOR assumptions of key bits based on the selection function in (7).
   - Find the XOR assumption with the greatest correlation value and it's the attack result for this row.
2. Repeat Step 1 to attack other rows and combine their results to recover the key bits, like in Example 2.

In summary, we can see that our method of attacking R_1 can recover all the key bits and this shows that an unprotected MAC-Keccak hardware implementations is vulnerable to side-channel attacks. This attack method is very different from the previous attacks on software implementations because software implementations have all the steps executed in serial and the intermediate results (*e.g.*, the compression step in θ and the output of θ) can be used to recover the key bits information directly. For hardware implementations, all five steps of a single round of MAC-Keccak are executed in one clock cycle and there is no register activity directly related to the intermediate key-dependent states of the linear operation output.

4 Discussion of Attacks and Countermeasures

In Section 3, we describe our method of attacking the first round output R_1 to recover the key bits. It is effective to attack hardware implementations of MAC-Keccak when the key size is 320. In this section, we discuss how to decrease the leakages of R_1 to protect MAC-Keccak against side-channel attacks, how the key length affects side-channel attacks, and the complexity of attacking Keccak compared to that of attacking other ciphers.

4.1 Discussion of Initial State in Keccak Implementations

This paper uses the traces sampled on one kind of implementation which resets the state register to all 0s before Keccak operation. Besides this kind of implementation, there are other commonly used implementations for Keccak, such as initializing the state register to the first round input P, or keeping the state register with the result of the last Keccak operation. For such situations, the Hamming distance model will be different.

If the state register is initialized to the first round input P, then the Hamming distance model for one bit will simply be:

$$HD(P(x_1, y_1, z_1), R_1(x_1, y_1, z_1))). \tag{8}$$

The bottom plane of P (K plane) contains all the 320 key bits while the top four planes of P is known by the attackers (message bits and padding bits). Thus attacking on bit $R_1(x_1, 0, z_1)$ in the bottom plane will involve another key bit $P(x_1, y_1, z_1)$ which makes the key guess a little more complex. Thus it is preferable to attack the bits (rows) in the tops four planes. In this case, attacking these 256 rows on the top four planes gives us a similar relationship to that we have found in the previous section - 1024 2-bit XORs with 320 distinct ones and 256 3-bit XORs:

$$\begin{cases} KG'_1 = \{S(x-1, 0, z) \oplus S(x, 0, z) \oplus S(x+1, 0, z-1)\}, \\ \qquad x \in \{1, 2, 3, 4\}, z \in \{0, 1 \cdots 63\}; \\ KG'_2 = \{S(x-1, 0, z) \oplus S(x+1, 0, z-1)\}, \\ \qquad x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\}. \end{cases}$$

Here the group KG'_2 is the same as KG_2 shown before, while the group KG'_1 is partial of KG_1. We plug in each of the 2-bit XORs in KG'_2 into a corresponding 3-bit XOR of KG'_1 and will recover 256 key bits first. These 256 bits will be all the key bits except for the 64 bits in $L_{0,0}$, i.e., $\{P(x, y, z),\ x = 0,\ y = 0\}$. We then use the recovered 256 key bits and plug them back into the 2-bit XORs KG'_2 to recover the remaining 64 key bits. Thus, we can see that the attack method we propose in this paper is also applicable to implementations with *Reg* initialized to other known values instead of 0.

For some other implementations with the state register keeps the result of last Keccak operation, the initial state of *Reg* is partially known by the attacker - the 256 bits of the output of the last Keccak operation. The known bits are $I([0:3], 0, [0:64])$ while they are in the first four lanes. According the the properties of ρ and π, by attacking these 256 bits, attackers can recover 192 different 2-bit XORs and 64 different 3-bit XORs and this can still leak a lot of information to attackers.

4.2 Protections of Keccak

As discussed in Section 4.1, the attackers will be able to extract some secret information if the initial state bits are known by the attackers. Thus the simplest countermeasure is to initialize the initial state to random numbers *rand* before the MAC-Keccak operations. In cryptographic systems, the random numbers *rand* are unknown to the attackers and this can help to decrease side-channel leakages. The only overhead of this method is from the random number generator.

To verify this assumption, we implement a 1600-bit random number generator based on a linear-feedback shift register (LFSR) and use the random numbers to initialize the state register Reg before the MAC-Keccak operations. We collect $400,000$ traces and run Pearson's correlation to look for correlation between $HW(R_1)$ and *power*. Results show that for such a simple protection method, the correlations between $HW(R_1)$ and *power* is only about 0.06, much smaller than the unprotected implementation (which reaches 0.5). The correlation is about 10 times less than the unprotected implementation, and therefore attackers will need about 100 times the number of traces to achieve the same success rate[13,14] as on unprotected implementation. We note here that this leakage is not decreased to 0 because combinational circuits activity will also incur leakages of R_1 and these leakages cannot be eliminated by this random number countermeasure.

Of course, initializing the state register with random numbers is a naive method which can help to decrease the leakages, not to eliminate all leakages. This method can be used as an add-on scheme together with other countermeasures. To totally eliminate the side-channel leakages, one currently used countermeasure is secret sharing protection [15,16]. Another simple countermeasure is to increase the length of key. We will discuss attacks for different key lengths in the next section.

4.3 Discussion about the Key Length

In this paper, we simplify the MAC-Keccak settings by assuming that the key length is 320 bits, which is just the size of one plane such that there will be no operation between key bits in the same column. Previous work [5] discusses the effect of the key length on side-channel attacks. Here we will summarize the effect of key length on side-channel attacks under our leakage model and attack method.

For *key-length* $\leq$ *plane-size*, the key bits will occupy no more than one plane and there is no operation between the key bits in the same column. We can attack on R_1 to recover all the key bits, as discussed in Section 3.

When *plane-size* $<$ *key-length* $\leq 2*$*plane-size*, the key bits occupy more than one plane. There will be operations between key bits from different planes. We can recover XORs of different numbers of key bits. For example, assume the key is 640 bits, thus the key bits take two planes P_0 and P_1. Then we can recover higher-dimension key bits XORs by attacking all the rows of R_1:

$$\begin{cases} KG_1^* = \{S(x-1,0,z) \oplus S(x-1,1,z) \oplus S(x,0,z) \\ \qquad \oplus S(x+1,0,z-1) \oplus S(x+1,1,z-1)\} \\ KG_2^* = \{S(x-1,0,z) \oplus S(x-1,1,z) \oplus S(x,1,z) \\ \qquad \oplus S(x+1,0,z-1) \oplus S(x+1,1,z-1)\} \\ KG_3^* = \{S(x-1,0,z) \oplus S(x-1,1,z) \\ \qquad \oplus S(x+1,0,z-1) \oplus S(x+1,1,z-1)\} \end{cases}$$

for $x \in \{0, 1 \cdots 4\}, z \in \{0, 1 \cdots 63\}$.

In this case, we can recover 320 5-bit XORs KG_1^*, 320 5-bit XORs KG_2^* and 320 different 4-bit XORs KG_3^* in total. Similar to the method used when the key size is no greater than 320 key bits, we can use all these XORs results to recover the entire K.

When *key-length* $> 2 \times$ *plane-size*, the key bits will occupy more than two planes and no complete rows contain only message bits. Attackers can attack bit by bit using DPA as discussed in Section 3.1.

4.4 How to Detect the Key Length

In the previous sections of this paper, we assume that the key length is known to the attackers, which does not hold in most applications. Usually, the attackers have no information about the length of the key used. Therefore it is a vital problem to get the key length for a black box MAC-Keccak system.

In [4], the authors propose to get the key length by performing differential power analysis (DPA). They start with the first message bit and repeatedly attack the next message bit until there is no correlation in a directly subsequent point in time. This method is effective because they target software implementations which have all the operations expanded and the intermediate variables can be used for attacks. For hardware systems, their method is no longer applicable and we should find a better method for hardware systems instead.

We denote the key length as l_k, the message length as l_m, and the appended length as l_a. Thus we have the following equation:

$$l_k + l_m + l_a = n_b * r \tag{9}$$

in which n_b is the number of blocks and r is the bit rate defined in Section 2.1. It is easy for the attackers to detect the number of blocks n_b, because this number will directly affect the length of operation, and the shape of the power waveform can also show the number of MAC-Keccak operations.

According to the appending rules of Keccak, $0 \leq l_a < r$, and this number varies according to the value of $l_k + l_m$ in which l_m is controllable by the attackers. Thus the strategy to get the key length is the following. Start from $l_m = 1$ and increase l_m, there exists a number α:

– For $l_m = \alpha$, the corresponding n_b is λ;
– For $l_m = \alpha + 1$, the corresponding n_b is $\lambda + 1$.

Then the key length is $\lambda * r - l_m$. Using this method, the attackers only needs to change the length of messages to detect the key length.

4.5 Discussion about the Attack Complexity

Our previous results show that unprotected hardware implementations of MAC-Keccak are vulnerable, and attackers can use the leakage of R_1 to recover all the key bits. However, compared to other cryptographic algorithms, MAC-Keccak is actually much more secure against side-channel attacks. For example, previous papers showed that for unprotected AES, the attackers need only several thousands traces to recover the key bytes [17]. For previous hash functions, like HMAC-based SHA-2 function, only 4000 traces are needed for successful side-channel attacks [18], while for unprotected MAC-Keccak, we need $35,000$ traces to get the five key bits operation results when attacking one row. When the key length is 320 bits, the attacker has to attack 320 rows to get the entire key, which is much more complex than previous cryptographic algorithms like AES and DES, where the number of attack iterations would be just 16 or 6.

In addition, previous block ciphers and hash functions based on them only have limited flexibility of key length, like AES-128 and AES-256. The attack methods for different key lengths are similar, using the divide-and-conquer approach. However, for MAC-Keccak, the key length is variable and the security level can be improved simply by increasing the key length, i.e., the difficulty for side-channel analysis will increase dramatically as well.

5 conclusion

In this paper, we propose a side-channel analysis method and power model specifically for hardware implementations of MAC-Keccak. Our method targets the first round output of Keccak and results show that our method is effective and efficient. Our work demonstrates that unprotected hardware implementations of MAC-Keccak are vulnerable to side-channel attacks. At the same time, our results show that compared to previous block ciphers and hash functions, MAC-Keccak is more resistant to side-channel attacks. In the future, we will investigate Keccak implementations resilient to side-channel attacks and develop more efficient countermeasures.

References

1. N. F. Pub, "DRAFT FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," *Federal Information Processing Standards Publication*, 2014.
2. G. Bertoni, J. Daemen, M. Peeters, and G. Assche, "The Keccak reference," *Submission to NIST (Round 3)*, January, 2011.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak sponge function family main document," *Submission to NIST (Round 2)*, 2009.
4. M. Zohner, M. Kasper, M. Stottinger, and S. Huss, "Side channel analysis of the SHA-3 finalists," in *Design, Automation Test in Europe (DATE)*, March 2012, pp. 1012–1017.
5. M. Taha and P. Schaumont, "Differential power analysis of MAC-Keccak at any key-length," in *International Workshop on Security*, Nov. 2013, pp. 68–82.
6. M. Taha and P. Schaumont, "Side-channel analysis of MAC-Keccak," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, June 2013, pp. 125–130.
7. P. Luo, Y. Fei, X. Fang, A. Ding, M. Leeser, and D. Kaeli, "Power analysis attack on hardware implementation of MAC-Keccak on FPGAs," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–7.
8. "Keccak hardware implementation in vhdl version 3.1," http://keccak.noekeon.org/KeccakVHDL-3.1.zip, 2014 (accessed May 14, 2014).
9. "Source codes for the SHA-3 round 3 candidates & SHA-2 - the third SHA-3 candidate conference release, March 2012," http://cryptography.gmu.edu/athena/.
10. "Tescase - testbed for side channel analysis and security evaluation," http://tescase.coe.neu.edu.
11. B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel, "Mutual information analysis," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, 2008, vol. 5154, pp. 426–442.
12. K. Kobayashi, J. Ikegami, S. Matsuo, K. Sakiyama, and K. Ohta, "Evaluation of hardware performance for the SHA-3 candidates using SASEBO-GII," Cryptology ePrint Archive, Report 2010/010, 2010.
13. Y. Fei, A. A. Ding, J. Lao, and L. Zhang, "A statistics-based fundamental model for side-channel attack analysis," Cryptology ePrint Archive, Report 2014/152, 2014.
14. A. Ding, L. Zhang, Y. Fei, and P. Luo, "A statistical model for higher order DPA on masked devices," in *Cryptographic Hardware and Embedded Systems – CHES 2014*, 2014, vol. 8731, pp. 147–169.
15. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Building power analysis resistant implementations of Keccak," in *Second SHA-3 Candidate Conference*, 2010.
16. B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. Van Assche, "Efficient and first-order DPA resistant implementations of Keccak," in *Smart Card Research and Advanced Applications*, 2014, pp. 187–199.
17. E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, 2004, vol. 3156, pp. 16–29.
18. R. McEvoy, M. Tunstall, C. Murphy, and W. Marnane, "Differential power analysis of HMAC based on SHA-2, and countermeasures," in *workshop on Information Security Applications*, 2007, pp. 317–332.