Side-Channel Analysis of MAC-Keccak

Mostafa Taha and Patrick Schaumont
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061, USA
Email: mtaha, schaum@vt.edu

***Abstract*—NIST recently completed the SHA-3 competition with the selection of Keccak as the new standard for cryptographic hashing. In this paper, we present a comprehensive Side-Channel Analysis of Keccak, when it is used with a secret key to generate a Message Authentication Code (MAC) (MAC-Keccak). Our analysis covers all the variations of the algorithm. We show that the side-channel resistance of the MAC-Keccak depends on the key-length used, and we derive the optimum key-length as $((n * rate) - 1)$, where $(n \in [2 : \infty])$ and $rate$ is the Keccak input block size. Finally, the paper demonstrates the feasibility of our side-channel analysis with a practical attack against MAC-Keccak implemented on a 32-bit Microblaze processor.**

I. Introduction

A hash function is a one-way function that converts arbitrary length messages into fixed-length digests. Hash functions are generally used in cryptography to ensure the integrity of a message. If the received message generates the same digest as the original one, the message should be correct, can never be altered. The United States National Institute of Standards and Technology (NIST) recently concluded a competition to select a new successor standard for SHA-1 and SHA-2. Keccak [1] was selected for the new standard SHA-3.

When hashing the combination of a secret key and a message, a Message Authentication Code (MAC) is obtained. A MAC ensures integrity as well as authenticity of the message, as the verification of a MAC implies testing knowledge of the secret key. However, MACs can be computed in several alternative forms, depending on how the message and the secret key are combined. In this paper, we will study the popular hash-based MAC (HMAC) [2] and the MAC construction recommended for Keccak [3] (MAC-Keccak).

We are specifically interested in side-channel analysis (SCA) of the MAC computation. Indeed, by exploiting side-channel leakage, the adversary may attempt to extract the secret key, which enables her to forge a MAC. Even partial disclosure of the internal, secret state of a hash algorithm may lead to forged messages. In this paper, we will study the side-channel leakage properties of MAC-Keccak, and we will show how the parameters of the algorithm can be configured to minimize harmful side-channel leakage.

The SCA of some of the previous SHA-3 candidates was studied in [4], [5] and [6]. Only Zohner et al describe SCA of MAC-Keccak [7]. They presented an analysis step based on the Correlation Power Analysis (CPA) [8] to identify the key-length used. They also proposed using the XORs of the θ step (discussed later) as the targeted point in the attack. In this paper, we will show that in most cases of MAC-Keccak, attacking the θ step is insufficient for a complete key recovery. We therefore take the analysis of Zohner et al one step further. We start from their method for identifying the key-length and assume that the key-length is variable but known upfront. We then do a comprehensive analysis of the effect of different key-lengths, which leads to the so-called optimum key-length: the length of a MAC-Keccak key that results in the most difficult side-channel analysis. We also study all possible attack points in the algorithm and address complete attack scenarios. Finally, we present the results of practical attacks against MAC-Keccak implemented in 32-bit Microblaze processor.

This research was supported in part by the VT-MENA program of Egypt, and by National Science Foundation Grant no. 1115839.

The paper is organized as follows. Section II presents several preliminaries on Keccak and MAC calculation. The analysis of MAC-Keccak is presented in Section III. The practical Attack and its results is presented in Section IV. The paper concludes in Section V.

II. Background

For completeness, we introduce a brief description of the Keccak hashing algorithm and the targeted MAC constructions. The discussion in this section is meant to be simple and intuitive. Further details can be consulted in [1].

A. Keccak

Keccak is a family of functions with a *sponge* construction, which is a generalized hash function that accepts arbitrary length messages and generates digests of any desired size. Keccak has several parameters:

- The state-size b, which is the size in bits of the internal state of the sponge. In Keccak, $b = 25 * 2^l$ and $l \in [0 : 6]$. $l = 6$ and $b = 1600$ are the default values.
- The rate r, which is the number of message bits that the sponge absorbs in every run (the input block).
- The capacity c, which is the number of zero bits that are appended to every r bits of the message to form the input state. The state-size equals the rate plus the capacity $(b = r + c)$.
- The output length o, which is the size of the required digest in bits.

The rate, capacity and the output length can be set to any value depending on the required security level. To match the NIST output length requirements, the designers of Keccak proposed

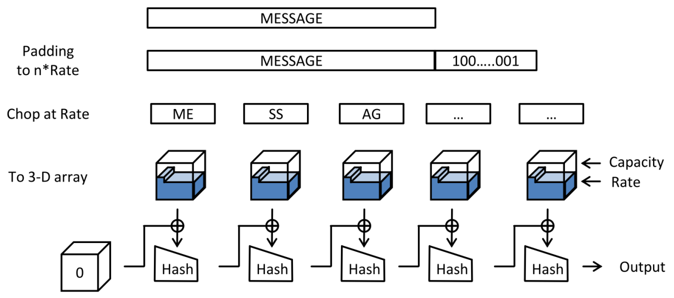


Fig. 1. Keccak hash algorithm and the padding rules

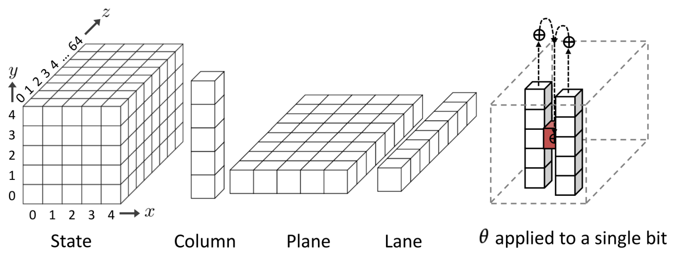


Fig. 2. (Partially from [1]), Keccak construction and the operation of θ

$r = 1152$, 1088, 832 and 576 for output length of 224, 256, 384 and 512 respectively [9]. The default rate for arbitrary length output is $r = 1024$.

The hashing is done in three steps:

Padding and Initializing : The input message is appended with a number of bits so that the total length is multiple of the rate. The padding starts with '1' and ends with '1' with all 0's in-between. Padding is mandatory, and therefore the minimum padding length is 2 and the maximum length is $(r+1)$. Padding is shown in the first step of Fig. 1. Initialization is done by setting the initial state to all zeros.

Absorbing : Every r-bit message bits (block) are appended with c (capacity) bits of zeros to be of the same size as the state. Then, the bits get arranged in a $(5 * 5 * 2^l)$ 3-D array form (called the state) starting from $(x = 0, y = 0, z = 0)$ and filling in the z direction, then x direction, then y direction. This filling sequence puts the new message bits in the lower planes (from $y = 0$) leaving the zero bits at the upper planes, which is indicated by the shaded part in each state of Fig. 1. The state is XORed with the previous state. The result is used as the input to the Keccak function, which outputs a new state. The absorbing operation continues for every block of message bits as shown in the figure.

Squeezing : The digest is the first o bits of the output state.

The Keccak function (represented by 'Hash' in the figure) consists of n_r rounds of five sequential operations. The number of rounds depends on the value of l used to define the state size with $n_r = 12 + 2l$. A single round of Keccak is represented as follows:

$$Output = \iota \circ \chi \circ \pi \circ \rho \circ \theta(Input) \tag{1}$$

We will describe these operations briefly:

- θ is a binary XOR operation with 11 inputs and a single output. Every bit of the output state is the result of XOR between itself, and the bits of two neighbor columns as shown in Fig. 2, more precisely:

$$s[x][y][z] = s[x][y][z] \oplus \left(\oplus_{i=0}^{4} s[x-1][i][z]\right) \oplus \left(\oplus_{i=0}^{4} s[x+1][i][z-1]\right) \tag{2}$$

- ρ and π are simple permutations over the bits of the state.
- χ is a mix between XOR, AND and NOT binary operations. Every bit of the output state is the result of XOR between itself and the AND between one neighbor and the NOT of another neighbor, more precisely:

$$s[x][y][z] = s[x][y][z] \oplus \left(\overline{s[x+1][y][z]} \cdot s[x+2][y][z]\right) \tag{3}$$

- ι is a binary XOR with a round constant.

B. HMAC and MAC-Keccak

A widely known MAC construction is the HMAC [2]. HMAC accepts a message M and a key K and generates the digest as following:

$$HMAC(M, K) = H((K \oplus opad)||H((K \oplus ipad)||M)) \tag{4}$$

Simply speaking, HMAC prepares a key variant $(K \oplus ipad)$ of the same size as the hash input block. It prepends the key variant to the message and hashes the resulting block. At the output, HMAC prepends the digest with another key variant $(K \oplus opad)$ and hash them one more time. The output key-padding in HMAC eliminates some security vulnerabilities related to retuning the hash state directly as the output. In Sponge constructions however, the output is only a small part of the final state. Hence, Keccak is protected by design from such vulnerabilities and we can use the direct MAC construction:

$$MAC(M, K) = H(K||M) \tag{5}$$

This method is recommended by Keccak designers and the recommended output length should be smaller than the capacity [3]. We will refer to the first method by HMAC, and to the second method by MAC-Keccak.

The goal of the adversary is to generate the correct digest without the knowledge of the legitimate key. For this purpose, the adversary can recover the actual secret key (Key Recovery), or recover the hash state after processing the key (MAC Forgery). In HMAC, MAC Forgery is the typical target. However, in MAC-Keccak we will need either of them depending on the key-length as will be discussed later.

III. SCA of MAC-Keccak

Keccak is a flexible algorithm with several parameters. Moreover, the MAC-Keccak does not specify a fixed key-length which adds more flexibility to the algorithm. In this section, we will analyze the MAC-Keccak algorithm from SCA point of view. We will start with the effect of key-length on the difficulty of the attack. Then, we will study the possible attack points and the leakage of each point. Finally, we will present a complete attack strategy. The analysis in this section is generic to any rate and capacity parameters.

A. *Effect of changing the key-length*

To study the effect of changing the key-length, we define the following quantities:

Number of Controlled Bits : The number of message bits in the input block that are variable and known to the adversary.

Number of Unknown Bits : The number of unknown bits that are required to forge a MAC digest. Depending on the key-length, this can match the number of bits in the key or the number of bits in the hash state.

Attack Difficulty : The ratio of the number of unknown bits over the number of controlled bits.

We will assume that the key-length can be any number of bits greater than zero. Small lengths are an easy target for brute force, but there is no fixed minimum length. So, we will keep it general to study the problem from SCA point of view. For this analysis, we will assume that the total number of message bits is larger than the rate. In other words, whatever the key-length was, the message will fill up the rest of the input block with no effect of padding bits. This is a very reasonable assumption in typical applications of cryptographic hashing. There are two cases of the key-length (l_K):

key-length $<$ *rate* : In this case, the key will be part of the input block and the message will fill up the rest of the block. The number of unknown bits will be the key-length. The process of extracting these bits is called Key Recovery. The number of controlled bits will be the rate minus the key-length ($r - l_K$). The attack difficulty will be ($l_K/(r - l_K)$) ranging from ($1/(r-1)$) to ($(r-1)/1$.

key-length $\geq$ *rate* : In this case, the key will fill up the first input block and part of the second block. The message will fill up the rest of the second input block. The number of unknown bits will be size of the previous state (b). Note that the recovery of the rest of key bits in the second input block is not required where; the key bits in the second block will be XORed with the previous state. The knowledge of the output of the XOR operation is sufficient to mount a MAC Forgery. Hence, the number of unknown bits will not increase beyond the state-size. The process of extracting these unknown bits is called MAC Forgery, as the adversary is not required to recover the original key. He will only recover the required information to forge a MAC digest. The number of controlled bits will be the rate minus the key-length modulus the rate ($r - (l_K (\text{mod } r))$). The attack difficulty will be ($b/(r - (l_K(\text{mod } r)))$) ranging from ($b/r$) to ($b/1$).

Figure 3 shows the number of controlled bits and the number of unknown bits as a function of the key-length. The number of controlled bits in every block decreases by increasing the key-length where the sum of them will always be the rate. The number of unknown bits will be equal to the key-length while the key-length is less than the rate. Once the key fills up a complete input block ($l_K = r$), the required unknown will be the previous state and the number of required unknown bits will be fixed at the state-size.

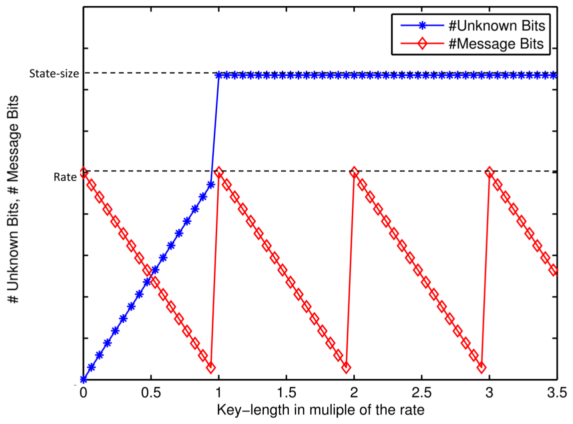


Fig. 3. Effect of key-length on the number of controlled bits and the number of unknown bits

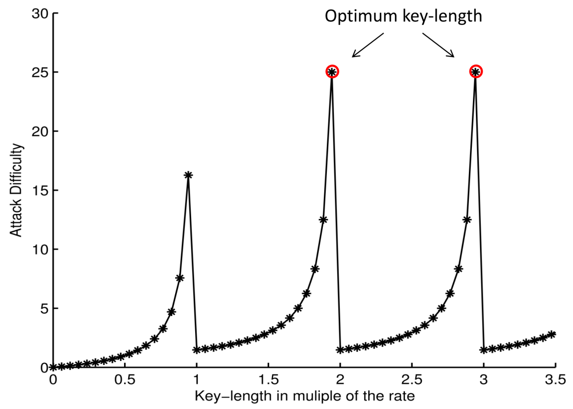


Fig. 4. Effect of key-length on the attack difficulty

Figure 4 shows the attack difficulty as a function of the key-length. The figure also shows that the optimum key-length from SCA point of view is ($(n * r) - 1$), where ($n \in [2 : \infty]$). Any increase of the key-length beyond ($(2 * r) - 1$) does not increase the security from SCA point of view.

From SCA point of view, HMAC-Keccak is a special case of MAC-Keccak when key-length match the rate. In the rest of the paper, we will focus on MAC-Keccak as the general case.

B. *Targeted Operation*

The Targeted Operation is an internal operation running within the module, where the adversary searches for its signature in the recorded leakage. The Targeted Operation must depend on both the controlled bits and the key as the adversary will search for the key that correctly links the change in the controlled input to the change in the power consumption. Hence, we assume that the adversary targets the first hash operation that manipulates the message. That is the first hash operation if ($l_K < r$) and the $n + 1$ hash operation if ($l_K \geq n * r$).

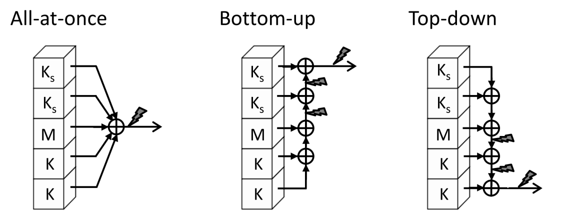


Fig. 5. Column XOR of θ operation

In this paper, we assume that the key prepends the message. The analysis can be easily ported to cases where the key is appended to the message. Following that assumption and that the state filling is bottom-up, the key bits will be in the lower planes and the controlled bits will be in the middle planes. If the key-length is less than the rate, the upper planes will be filled with zeros. Otherwise, the upper planes will be filled with the previous state (key-state).

Due to the flexibility of the key-length in MAC-Keccak, no single targeted operation will be sufficient for all the cases. We will study all the targeted operations that will be sufficient to mount an attack in any key-length. The operations are studied in the same order of the algorithm. The success of attacking a later operation depends on the success of attacking the previous operations, as the previous results will be used as assumptions in later attacks.

State XOR : The XOR operation between the new input block and the previous state is the first targeted operation. This operation is useful only if the ($l_K \geq r$), where the previous state is the required unknown. In this case, the State XOR operation can recover the bits that are XORed with the controlled bits ($r - (l_K (\text{mod } r))$) out of the State-Size bits (b). The number of bits that are XORed in parallel depends on the implementation ranging from all-at-once in hardware modules (FPGAs, and ASICs) to the processor width in software modules.

The θ operation : Understanding the implementation of this step is critical in SCA. Typically, θ is done in two steps, first the module calculates the XOR between the elements of each column (five elements) and generates what can be called θ_{plane}. Then, it calculates the XOR between every element in the state, and two neighbor locations of the θ_{plane} as indicated earlier (Fig. 2).

The first step of θ : The analysis of the first operation (generation of the θ_{plane}) depends on the order of XORing the five elements as shown in Fig. 5 where we assumed that the key-length is greater than the rate as a general case. The key-state bits (K_s) can be replaced by 0's otherwise. Generation of the θ_{plane} can be done all-at-once in hardware modules (FPGAs, and ASICs), or bottom-up or top-down in software modules, where the first method (bottom-up) is used in the reference software implementation [10].

- In all-at-once implementations, the output of the gate will take the effect of all the unknown bits at once. Hence, SCA cannot recover the value of individual unknown bits however; it can only get the result of XORing all of them. Targeting this operation can be a complete attack only if there is one unknown bit in every column, that there will be only one unknown bit in every operation i.e. $l_K \leq p$; where p is the size of one plane (320 bits).
- In bottom-up implementations, all key bits will be XORed with each other before being XORed with a controlled bit. Then, the result will be XORed with key-state (K_s) bits (if there are any) in sequential steps. Similarly, SCA can only reveal the result of XORing all the key bits. However in this implementation, SCA can recover all the key-state bits by monitoring the result of each XOR operation (recover one key-state bit at every operation). Targeting this operation can be a complete attack only if there is only one key bit in every column ($l_K(\text{mod } r) \leq p$).
- In top-down implementations, the situation is exactly reversed from that in bottom-up implementations. SCA can only reveal the result of XORing all the key-state bits and it can recover the individual values of all the key bits. The role of bottom-up and top-down will be switched if MAC construction is built with the key appending the message.

Figure 5 shows the possible attack points in every implementation. In every case discussed above, if there is at least one controlled bit in every column ($r - l_K(\text{mod } r) \geq p$), the targeted operation will be enough to get the θ_{plane} which is a valuable information for attacking the second step of θ.

The second step of θ : In this step, the algorithm calculates the XOR between every element of the state and two neighbor elements of the θ_{plane}. If the θ_{plane} is correctly found in the previous step, SCA can recover all the unknown elements of the state.

To conclude, the two steps of θ can be used to build a complete attack only if there is at least one controlled bit in every column. For smaller number of controlled bits, the adversary will have to follow the Keccak function in the later operations.

ρ and π operations : These permutation operations cannot be used as an SCA target because they do not involve any mixing between known and unknown parts.

The χ operation : This operation is a good SCA target. Every bit involved in this operation carries information from 11 controlled or unknown bits. Since it is a 3 input operation, the output will depend on 33 controlled or unknown bits.

The ι operation : Similar to permutation operations, this operation does not reveal any SCA leakage because it does not involve any mixing between known and unknown parts.

Operation Width : One last note in the analysis of the targeted operations is the width of the operation (the number of parallel bits included in the SCA). All Keccak functions are binary operations, which makes the adversary free in choosing the suitable width i.e. he can build the CPA software to search for 1-unknown bit at a time, 2-unknown bits or 8-unknown bits (byte) etc... However, the choice of the operation width is a trade-off. If the adversary chooses a one bit width, he will get

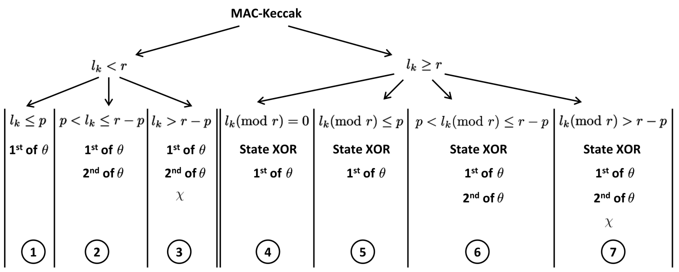

MAC-Keccak
$l_k < r$
$l_k \geq r$
$l_k \leq p$
$p < l_k \leq r - p$
$l_k > r - p$
$l_k (\text{mod } r) = 0$
$l_k (\text{mod } r) \leq p$
$p < l_k (\text{mod } r) \leq r - p$
$l_k (\text{mod } r) > r - p$
1st of θ
1st of θ 2nd of θ
1st of θ 2nd of θ χ
State XOR 1st of θ
State XOR 1st of θ
State XOR 1st of θ 2nd of θ
State XOR 1st of θ 2nd of θ χ
1 2 3 4 5 6 7

Fig. 6. Complete Attack Scenarios

a small search space of only $2^1 = 2$ but increased algorithmic noise where the power consumption of all the other parallel bits will contribute to noise. On the other hand, if the adversary targets all the parallel bits at once (higher operation width), he will get zero algorithmic noise, but the search space will be huge 2 to the power of number of parallel bits.

C. Complete Attack Scenarios

After analyzing the elements of MAC-Keccak, now it is the time to glue the previous two sections together in complete attack scenarios. Fig. 6 shows all possible variations of the key-length and the required attack steps in each case where p in the figure refers to the size of one plane. We assume that the first step of θ is implemented bottom-up.

Key-length less than the rate :

1) $l_K \leq p$: There is at-most one key bit in every column. The first step of θ is enough to complete the attack.
2) $p < l_K \leq (r - p)$: There are two or more key bits and at least one controlled bit in every column. The first step of θ will be used to get the θ_{plane}, and the second step of θ will be used to get the values of individual key bits.
3) $l_K > (r - p)$: There are some columns without any controlled bit. θ operation will not be enough, and the adversary will have to track down the χ operation.

Key-length greater or equal to the rate :

4) $l_K(mod\ r) = 0$: The State-XOR will be used to reveal some bits of the key-state (equal to the rate). The first step of θ will be used to get the rest of key-state.
5) $l_K(mod\ r) \leq p$: The State-XOR will be used first to get some key-state bits (equal to the controlled bits). The first step of θ will be used to get the key bits and the rest of key-state bits.
6) $plane < l_K(mod\ r) \leq (r - p)$: There is at least one controlled bit in every column. The State-XOR and θ will be enough.
7) $l_K(mod\ r) > (r - p)$: There are some columns without any controlled bits. The State-XOR, θ and χ will be needed.

In the following section, we will present two insights in the analysis of θ operation. Then, we will show results for a complete attack.

IV. Practical Attacks

In the practical experiments, we used the reference software implementation of Keccak [10] on a 32-bit Microblaze proces-

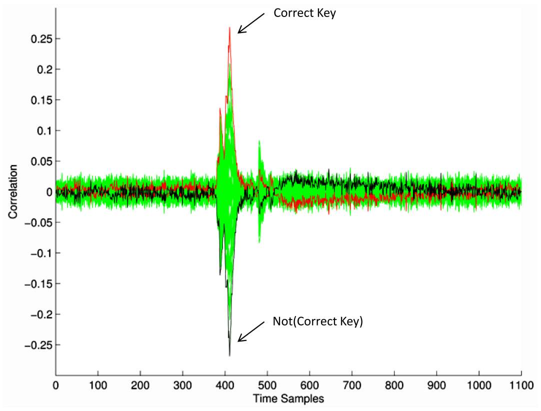


Fig. 7. SCA of an XOR operation

sor [11] running on top of a Xilinx Spartan-3e FPGA. We used a Tektronix CT-2 current probe as an indication of the power consumption. In the analysis, we used the Hamming Weight power model. We build the power model for 8-bit of the key at a time which is a reasonable choice between increased search space and increased algorithmic noise. We used the Pearson Correlation Coefficient [8] to measure the similarity between the actual and the modeled power consumption.

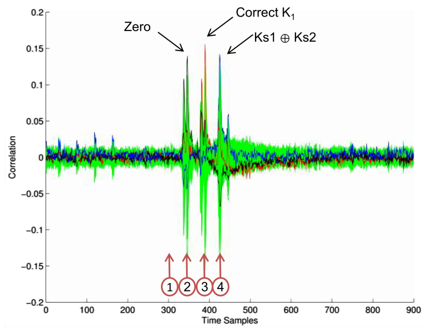
A. SCA of XOR operation

Before conducting a complete attack, we first start with analyzing the side-channel leakage of one XOR operation. We targeted the last XOR operation of generating the θ_{plane} with 50,000 power traces. Figure 7 shows the result of this analysis. The red plot is for the correct key guess, the black plot is for the complement of the correct key and the green plots are for all other key guesses.

The figure clearly shows the effect of attacking simple XOR binary operations. If the key is miss-guessed by one bit, the result power model will differ for only one bit. This is the reason for the high correlation in many of the green plots. In order to get over the high correlation of incorrect key guesses, the adversary needs to collect increased number of power traces. The correct key guess shows up clearly after around 10,000 traces. Moreover, if the key is guessed to the complement of the correct key, the result will show a strong negative correlation. This means that, the adversary needs to consider only the positive correlation.

B. SCA of generating the θ_{plane}

Another interesting observation found in the analysis of generating the θ_{plane} is that the peak positive correlation is always found in triple spikes, as shown in Fig. 8. The reason of this can be explained with the help of the XOR operation on the right hand side of the figure. We first assume that the key-length equals to the rate. The required unknown in this case will be the entire hash state before processing the message. We further assume that the key-bits that are directly XORed with the input message in the State-XOR operation could be recovered successfully. This means that, the first three planes



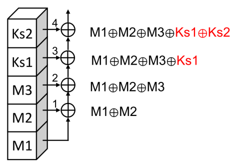


Fig. 8. SCA of generating the θ_{plane} operation

TABLE I
PRACTICAL EXPERIMENTS

Key-length in bits	320 1 plane	640 2 planes	1088 rate	1408 rate+1 plane	1728 rate+2 planes
Scenario	1	2	4	5	6

will be filled with known variables while the upper two planes are filled with the unknown previous key-state. The attack is meant to recover the first unknown (Ks1). The correct key-state bit at the output of the third XOR is the required unknown (Ks1) which shows up as the middle spike. Keeping in mind that, the power traces show all the operations sequentially. The second XOR operation shows up at the left spike with high correlation at a false key '0'. Similarly, the fourth XOR operation shows up at the right spike with high correlation at a false key Ks1 $\oplus$ Ks2. This observation is critical for a correct attack. The three results represent actual operations running within the module that, the three correlation spikes should be of the same value. If the CPA is applied blindly without time profiling, the best guess will be uniformly distributed between '0', the correct key-state bit, and the XOR of the rest of key-state bits. Based on this observation, we used a precise time profiling to target only the intended operation.

C. Complete Attack

The practical experiments was conducted with these Keccak parameters: state-size ($b = 1600$), rate ($r = 1088$) capacity (c = 512). In this configuration, the plane size ($p = 320$), and the input block covers 3 planes and 2 lanes. We tested five cases with different key-lengths as shown in Table I. We chose these cases to cover all the scenarios discussed in Fig. 6 except for those that requires χ operation (case 3 and 7) which we will leave for future work. The steps involved in the analysis of each case can be recalled from Fig. 6.

Figure 9 shows the success rate of each experiment as a function of the number of traces. The figure clearly validates the effect of changing the key-length on the SCA difficulty. For a fixed number of attack traces (e.g 15,000 traces), every case will achieve a different success rate. The *1 Plane* case involved only one SCA task (1^{st} of θ), and achieved the highest success rate. The *Rate + 2 Planes* case involved three SCA tasks (State-XOR, 1^{st} of θ and 2^{nd} of θ) and achieved the worst success rate.

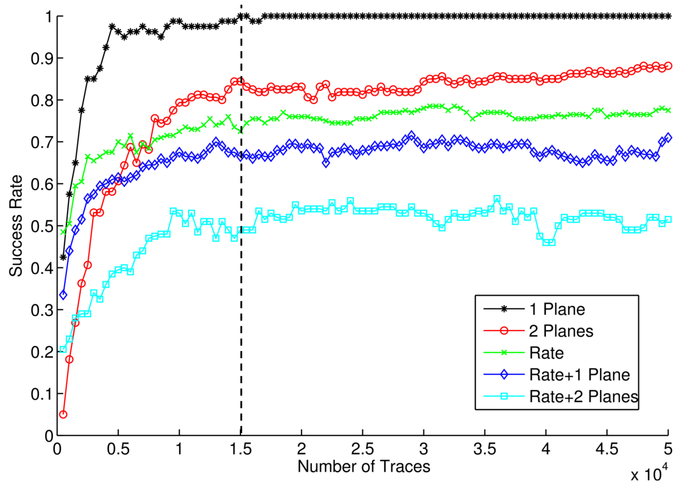


Fig. 9. The effect of changing the key-length on the SCA success rate

V. CONCLUSION

In this paper we presented a comprehensive analysis for Keccak hash algorithm in MAC construction. We presented analysis of the effect of changing the key-length, all the possible attack points and complete attack scenarios. We supported the analysis with practical experiments on a 32-bit Microblaze processor. The paper concluded that the optimum key-length from SCA point of view is $((n * rate) - 1)$, where $(n \in [2 : \infty])$.

REFERENCES

[1] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The keccak reference," *Submission to NIST (Round 3)*, vol. 3.0, 2011. [Online]. Available: http://keccak.noekeon.org/Keccak-reference-3.0.pdf
[2] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," in *Advances in Cryptology CRYPTO 96*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1996, vol. 1109, pp. 1–15.
[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Cryptographic sponge functions," vol. 0.1, 2011. [Online]. Available: http://sponge.noekeon.org/CSF-0.1.pdf
[4] P. Gauravaram and K. Okeya, "Side channel analysis of some hash based MACs: a response to SHA-3 requirements," in *Information and Communications Security*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5308, pp. 111–127.
[5] O. Benoît and T. Peyrin, "Side-channel analysis of six SHA-3 candidates," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6225, pp. 140–157.
[6] C. Boura, S. Lévêque, and D. Vigilant, "Side-channel analysis of Grøstl and Skein," in *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, May 2012, pp. 16 –26.
[7] M. Zohner, M. Kasper, M. Stöttinger, and S. Huss, "Side channel analysis of the SHA-3 finalists," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, Mar. 2012, pp. 1012 –1017.
[8] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3156, pp. 135–152.
[9] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The keccak sha-3 submission," *Submission to NIST (Round 3)*, 2011.
[10] "Keccak reference code submission to NIST (round 3)," http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip.
[11] "Xilinx microblaze soft processor core," http://www.xilinx.com/tools/microblaze.htm.