Towards Secure Cryptographic Software Implementation Against Side-Channel Power Analysis Attacks

Pei Luo*, Liwei Zhang† Yunsi Fei*, A. Adam Ding†
*Electrical & Computer Engineering Department, Northeastern University, Boston, MA 02115 USA
†Department of Mathematics, Northeastern University, Boston, MA 02115 USA
silenceluo@coe.neu.edu, zhang.liw@husky.neu.edu, yfei@ece.neu.edu, A.ding@neu.edu

***Abstract*—Side-channel attacks have been a real threat against many embedded cryptographic systems. A commonly used algorithmic countermeasure, random masking, incurs large execution delay and resource overhead. The other countermeasure, operation shuffling or permutation, can mitigate side-channel leakage effectively with minimal overhead. In this paper, we target automatically implementing operation shuffling in cryptographic algorithms to resist against side-channel power analysis attacks. We design a tool to detect independence among statements at the source code level and devise an algorithm for automatic operation shuffling. We test our algorithm on the new SHA3 standard, Keccak. Results show that the tool effectively implements operation-shuffling to reduce the side-channel leakage significantly, and therefore can guide automatic secure cryptographic software implementations against differential power analysis attacks.**

***Keywords*—Side-channel attacks, shuffling, Keccak**

I. INTRODUCTION

Side-channel analysis (SCA) has been an effective and practical attack on many critical embedded systems that employ cryptographic algorithms for security [1]. SCA exploits the correlation between physical leakages of a crypto system and its secret key-dependent intermediate variables to retrieve the key. Various countermeasures have been presented to protect embedded systems from side-channel analysis. Secret sharing (random masking) [2] introduces random numbers into the system to mask the secret key. It requires algorithm-specific modifications which are hard to automate, and normally incurs large implementation overhead as each share needs one copy of the original algorithm.

Other countermeasures such as random delay [3] and shuffling [4] have also been studied. These methods spread the side-channel leakage (correlated to an intermediate variable) from a single time point onto multiple points so as to decrease the leakage. In [4], [5], random permutations are applied on AES operations to resist first-order differential power analysis (DPA), based on the fact that AES is a block cipher where the 16 operations on different key bytes and state bytes in each round are independent. In [6], a simplified version of shuffling, Random Start Index (RSI), is presented. RSI is easier to implement but can be significantly weaker than the random permutation method.

Compared to masking, shuffling does not require modifications of the algorithm. It is an algorithm-agnostic implementation and can possibly be automated for any cryptographic algorithms. What's more, it can be easily implemented after other countermeasures as an add-on protection for cryptographyic systems. However, manual implementation of shuffling still requires knowledge of the specific algorithm and may not fully exploit the independence between operations in complex algorithms. Recent works [7], [8], [9], [10], [11] indicate a nascent trend towards automating the application of countermeasures against SCA to increase the security of systems. They have focused on masking AES, including automatic instruction sensitivity quantification and local random precharging [7], a general code morphing engine design with alternative code segments that mitigate power leakage [8], compiler assisted masking implementation [9], and automatic security evaluation and verification [10], [11]. However, to the best of our knowledge, there is no automation work for operation shuffling/permutation yet.

In this paper, we propose a methodology to analyze the source code of crypto algorithms to automatically detect the dependence of statements. We then devise an algorithm to implement shuffling automatically at the source code level. We start from the high level, source code, as shuffling statements at this level are both algorithm and platform independent. We test our algorithm on Keccak, the new SHA3 standard. The experimental results show that our algorithm automatically identifies independence among operations and implements shuffling, which significantly improves the resilience of crypto software against power analysis attacks. The main contribution of the work lies in a framework of source code transformation to randomize operations without any knowledge of the underlying systems, which is generally applicable to other cryptographic algorithms as well.

The rest of this paper is organized as follows. Section II introduces some preliminaries on shuffling. In Section III, we present our algorithm to extract the dependence among software statements and propose a methodology for automatic shuffling implementation. In Section IV, we demonstrate correlation power analysis results on Keccak protected with shuffling based on our algorithm. We conclude the paper in Section V.

II. PRELIMINARIES

A. Countermeasure to Power Analysis Attacks - Shuffling

Shuffling is an effective countermeasure to mitigate the vulnerability of cryptographic systems against power analysis. As the leakage comes from intermediate variables produced by operations, if there are n_p possible time locations of a leaky operation, i.e., the shuffling space is n_p, the leakage of such intermediate variable is randomly spread onto n_p time points. The corresponding correlation and signal-noise-ratio

(SNR) for power analysis attacks will be effectively decreased to $1/n_p$ [4], [5].

When one piece of code C is composed of N independent segments with some of them producing key-dependent intermediate variables, $C = \{C_0, C_1, \cdots, C_{N-1}\}$, they can be executed in any order without changing the algorithm functionality. Their execution order is determined by an array, Order, which is a random permutation of $\{0, 1, \cdots N-1\}$.

Shuffling implementation includes two critical steps, extraction of statement dependence and generation of the random execution order array Order. Fisher-Yates algorithm [12] is an efficient algorithm for generating such an unbiased array with linear computational complexity $O(n)$. It is an unbiased algorithm, which means the resulting Order array will be fully random [13]. Thus we adopt this algorithm for random order array generation and expect it to impose small execution overhead in large crypto systems.

B. Data Dependence and Operation Consistency

For shuffling, the most important step is to extract the dependence of statements to identify the space for shuffling. We first make some definitions for software source code:

- We denote a piece of code C as a set of sequential statements $A = \{A[0], A[1], \cdots, A[N-1]\}$. Here N is the number of statements, called code size. We assume the code is branch-free and loop-free, i.e., consisting of only sequential static single assignments (SSA).
- For each statement $A[i]$, it includes the variable set $V[i] = \{v_i[0], v_i[1], \cdots\}$, with $v_i[0]$ being its output and other variables being its inputs. Statements may have different numbers of input variables.
- For statements $A[i]$ and $A[j]$, with $i > j$, if there is dependence between the two statements, we say that $A[i]$ depends on $A[j]$, denoted as $A[i] \Rightarrow A[j]$.

There are three types of dependence between two sequential statements $A[i]$ and $A[j]$, $i > j$, defined in [14]:

- Read after write (RAW): if $A[i]$ uses a variable that was defined by $A[j]$, their execution order has to be preserved;
- Write after read (WAR): if $A[i]$ redefines a variable that was used by $A[j]$, their execution order has to be preserved;
- Write after write (WAW): if $A[i]$ redefines a variable that was defined by $A[j]$, their execution order has to be preserved. Some compilers may optimize this WAW dependence if no usage of the variable is found between the two statements. At the source code level, we keep such dependence.

Shuffling, although aims at changing the execution order of statements, should preserve their inherent dependence. We define this as the rule of operation consistency.

Example 2.1: Here we give a piece of code as an example, and denote it as a sequence of statements as following:

$$\begin{cases} \mathsf{a = b+c+d} \\ \mathsf{e = a+f} \\ \mathsf{a = g+h} \end{cases} \Longrightarrow \begin{cases} A[0]: & V[0] = \{a, b, c, d\} \\ A[1]: & V[1] = \{e, a, f\} \\ A[2]: & V[2] = \{a, g, h\} \end{cases}.$$

There exists RAW dependence between $A[1]$ and $A[0]$, WAR dependence between $A[2]$ and $A[1]$, and WAW dependence between $A[2]$ and $A[0]$. Therefore, we have $A[1] \Rightarrow A[0]$, $A[2] \Rightarrow A[1]$, and $A[2] \Rightarrow A[0]$. The order of $A[0]$, $A[1]$, $A[2]$ cannot be permuted because of these dependence.

III. Methodology and Algorithm

A. Statement Dependence Extraction

The simple Example 2.1 shows that the dependence determines the constraints for operation shuffling and should be extracted first. For a piece of code with N statements, we use an $N \times N$ matrix M to denote the dependence of these N statements, and it should only be lower-triangular according to the definition of data dependencies. With the row index i and the column index j, $M[i][j] = 1$ means that $A[i] \Rightarrow A[j]$, for $0 \le i, j \le N-1$, $j < i$. The algorithm for dependency matrix generation is shown in Algorithm 1.

Algorithm 1 Dependence extraction

Input: The source code of the crypto algorithm
Output: Lower triangular dependency matrix M for the algorithm

```
N ← number of statements |A|
Generate an N × N matrix M, initialized at zero
for i = 1 → N − 1 do
    for j = 0 → i − 1 do
        if V[j][0] ∈ V[i] then   ▷ A[i] ⇒ A[j], WAW and RAW
            M[i][j] ← 1
        end if
        if V[i][0] ∈ V[j] then   ▷ A[i] ⇒ A[j], WAW and WAR
            M[i][j] ← 1
        end if
    end for
end for
```

Using Algorithm 1, we can extract the dependence among all the statements. This dependency matrix helps designers automate shuffling. Finding a group of statements for shuffling is to find a sub-matrix in M along the main diagonal with all its elements zero, i.e., the corresponding statements are mutually independent.

We use the Keccak source code (Keccak-simple32BI [15]) as an example to apply Algorithm 1. There are 344 statements (SSAs) for the first two rounds (in one loop), and thus the dependency matrix size is 344×344. Fig. 1 shows a 36×36 dependency sub-matrix for the first 36 statements.

For the 36×36 matrix, the square (i, j) in black color means $M[i][j] = 1$, i.e., $A[i] \Rightarrow A[j]$. For these 36 statements, it is clear to see that $A[0]$ to $A[9]$ are mutually independent, and thus these 10 statements can be randomly permuted. This is also true for $A[10]$ to $A[19]$. Note that the sub-matrix does not need to be composed by consecutive rows and columns. For example, $\{A[21], A[23], A[25], A[27], A[29]\}$ and $\{A[20], A[22], A[24], A[26], A[28]\}$ can also be permuted as their corresponding sub-matrices are both zero. We also

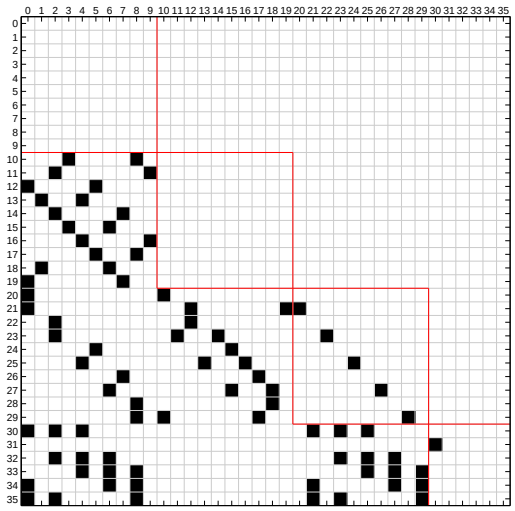

Fig. 1: The dependence of the first 36 statements of Keccak

analyze the AES code from [16] using our algorithm and similar dependence result is obtained. For larger cryptographic software, large N and complex dependence require automatic methods for dependence extraction and operation shuffling to be designed.

So far we have shown that shuffling is done on operations within basic blocks, i.e., relying on the data flow of basic blocks and exploiting the dependence. Shuffling can also be applied at the higher control flow level between basic blocks if the basic blocks are independent from each other. For example, the MixColumns step of an AES round consists of four basic blocks with each basic block working on one column of the AES state. These four blocks are mutually independent and can be permuted. However, the space for shuffling at the basic block level may be limited. To fully explore the shuffling space, we can apply multi-level permutations, where independent statements inside each basic block are permuted and meanwhile the serialization of basic block execution is also randomized (permuted).

B. Automatic Shuffling

With the dependence of statements extracted, we separate all the statements into different levels, such that the statements at the same level are mutually independent and can be permuted randomly without violating the rule of operation consistency. Statements at different levels have data dependence and should preserve a certain execution order: statements at the lower level should be executed before the higher level ones.

Algorithm 2 shows the procedure to separate all statements into different levels. It is actually running a reverse depth-first search on the data flow graphs of the program, and assigns the statements (nodes in the DFGs) to different depths.

We apply our algorithms on AES and Keccak, and the results show that our algorithms can effectively extract the dependence of statements and explore the shuffling space. For example, for the loop-free AES implementation, 55 statements are separated into 8 levels and the maximum number of

Algorithm 2 Execution level extraction

Input: The $N \times N$ dependency matrix M
Output: Execution level array $L[N]$

```
1: Initialize an zero array L[N]
2: for i = 1 → N − 1 do
3:     level = 0, newlevel = 0
4:     for j = 0 → i − 1 do
5:         if M[i][j] = 1 then
6:             newlevel = L[j] + 1
7:             if newlevel > level then
8:                 level = newlevel
9:             end if
10:        end if
11:    end for
12:    L[i] = level
13: end for
```

statements at a level is 16; for Keccak (Keccak-simple32BI), the 344 statements can be separated into 26 levels, with the maximum number of statements at a level as 50. When the number of statements in one level is too small, some dummy statements can be added into this level to increase the shuffling space but with a moderate performance penalty. Note here the independent statements may not all be leaky operations, i.e., producing key-dependent intermediate variables. They just represent data flow dependence. Only permutation of the leaky operations would decrease the side-channel leakage.

IV. Experimental Results and Evaluation

A. The Tool for Source Code Transformation

Our algorithms above operate on statements at the source code level without the need to understand the target cryptographic algorithm. Therefore, tools based on the proposed algorithms only need some lexical analysis, but do not need the complex semantic analysis that compilers have. In this work, we develop a source code transformation tool based on our algorithms using Python. The structure of our tool is shown in Fig. 2.

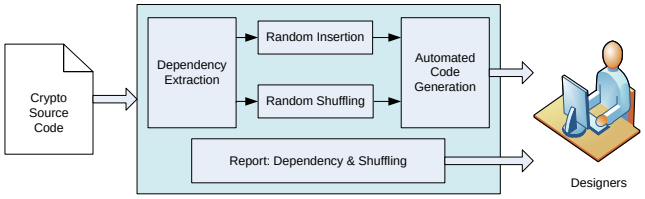

Fig. 2: Automatic source code transformation with shuffling

Our automatic source code transformation tool consists of the following steps for shuffling:

1) Lexical analysis: Read and parse the loop-free source code, extract all the variables in each statement. Lexical analysis is the first step for compilation process and there exist mature tools that can be applied, for example, we use FLEX [17] for test here.
2) Dependence analysis: Use Algorithm 1 shown in Section III to analyze the dependence among the statements in source code and generate the dependency matrix.

3) Space exploration: Use Algorithms 2 to explore the space for statement shuffling.
4) Code generation: Based on the shuffling space exploration results, embed a run-time permutation engine that contains code for generating the random execution order array, and generate a report about dependence and shuffling space at the same time.

As described before, only part of the independent statements leak secret information and permutation on them would reduce the leakage. If some knowledge about the secret information can be used in the "Random Shuffling" step, the implementation will be more efficient. For example, as most of the previous attacks on Keccak software implementation focus on θ step, we can apply the code transformation to this part of source code only and the implementation can be effective in reducing the leakage and still be efficient.

The designers only need to run this tool one time at the design stage. The generated code will contain a module that generates the random order array at run-time. The computational complexity of Algorithm 1 and 2 are both $O(n^2)$. For one round of Keccak which contains 344 statements, our Python implementation needs less than one second to finish all the work including dependence extraction and shuffling generation. Note that the transformations done by our tool will be orthogonal to normal compilation processes and will not be changed by later compilation. At runtime, a random array Order will be generated each time before the cryptographic execution, and the computational complexity of random array generation algorithm is $O(n)$. Details about the overhead on a target platform will be discussed in Section IV-C.

B. Side-Channel Leakage Reduction

To evaluate the effect of our automatic shuffling application, we run our algorithm on the source code of Keccak, Keccak-simple32BI, and generate an automatically shuffled version. We implement both the original and shuffled implementations on a 32-bit Microblaze processor running on an SASEBO-GII board with a Virtex-5 FPGA [18].

We use similar setting for Keccak as previous papers [19], [20], [21]. Without loss of generality, we assume the key size is 320 bits. For software implementations, there are many intermediate variables that can be used for side-channel analysis. We also attack the first step of θ which compresses every five bits (including one key bit) into one bit. With our dependence detection algorithm, statements of θ_1 are all separated into the first level which contains 10 statements. Meanwhile, the results of θ_1 are used in step θ_2 and they are separated to second level which also contains 10 statements. We only permute the two groups with each group containing 10 leaky statements. Thus each of the original two leakage points is spread onto 10 points and we expect to see a 10 times decrease of the leakage. For power analysis, we collect power traces for the two implementations using a LeCroy WaveRunner 640Zi oscilloscope. We first run CPA in time domain on the two MAC-Keccak implementations. The leakage model is Hamming weight of eight bits of θ_1, and the correlation results are shown in Fig. 3.

Fig. 3(a) shows the correlation between the Hamming weight of eight bits of θ_1 output and the power consumption

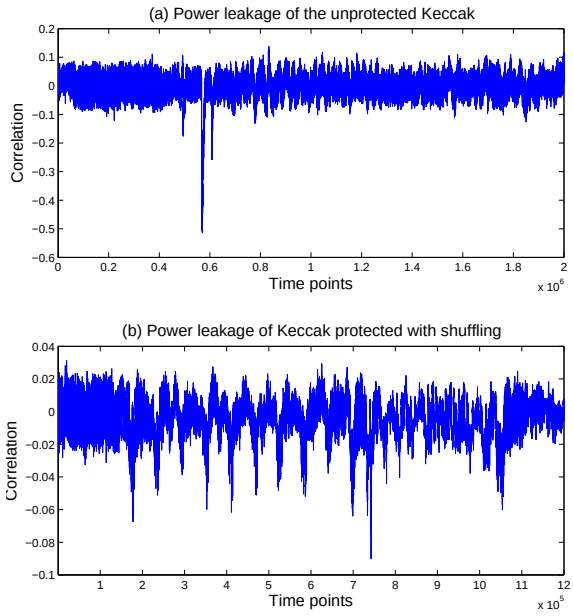

Fig. 3: Power leakage of the unprotected and shuffled Keccak on Microblaze

for the unprotected Keccak implementation based on $1,000$ power traces. With just $1,000$ traces for the unprotected MAC-Keccak, the peak correlations at the two leakage points (0.55 and 0.27) stand out from the noisy correlations at other non-leaky time points (varying between 0.1 and -0.1). Fig. 3(b) shows the correlation result of the shuffled implementation with 15,000 power traces. The peak correlation is much smaller, which implies that more power traces are required to reduce the variance of other noisy correlations. For the shuffled implementation, the two leakage points are spread onto twenty time points clearly. The average correlation of these points is close to 0.055, which is about $\frac{1}{10}$ of 0.55, the original correlation. The largest correlation is only about 0.09. With the decrease of correlation (SNR), by the formula in [22], the number of traces needed for protected implementation, to achieve the same success rate, will be 36 times of the number for unprotected implementation.

Note that Fig. 3 only shows the leakage reduction of one key byte. When groups of statements are permuted, since different statements involve different key bytes, the leakages on other key bytes are also reduced significantly. This will make the full key recovery much harder.

Previous works show that while some countermeasures like random delay can reduce the leakages in time domain, they are ineffective in frequency domain [23], [24]. Thus we also check if our operation shuffling algorithm can improve the side-channel security of embedded crypto systems in frequency domain. We find that in frequency domain, the SNR is very small and thus we use Hamming weight of 32 bits instead of 8 bits together for correlation analysis. For the unprotected MAC-Keccak implementation, the highest correlation at certain frequency is very clear and reaches 0.2. While for shuffled implementation, the largest correlation is very unclear and only about 0.05 instead. This result shows that shuffling can also effectively improve the resilience of the crypto system to side-

channel attacks in frequency domain.

C. Overhead Evaluation

Evaluating the execution time and resource overhead is important for countermeasure design because of the limited resources in embedded systems. The comparison of binary file size and clock cycles for three different MAC-Keccak implementations on Microblaze is shown in Table I. Note here the secret sharing scheme is a manual two-share masking implementation and one random number is involved for masking [25]. The shuffling implementation is the one that our automation tool generates.

TABLE I: Comparison of resources and execution cycles among three schemes

Implementations	File size (Byte)	Clock cycles
Original [15]	31040	1670
Shuffling	40128	2580
Secret Sharing [25]	69272	6780

All three implementations are implemented in plain C language on the same platform with the same compilation and measurement settings. The second column of Table I gives the binary file size of each implementation and the third column shows the execution clock cycles. We can see that shuffling incurs 29.3% memory overhead and the execution time is 1.54X compared to the unprotected version. For secrete sharing, the memory overhead is 123% and the execution time is 4.06X. This is because secret sharing involves share generation and de-masking and this needs more resources than other schemes.

From the overhead results and the analysis above, we can see that our algorithm is effective to automate shuffling implementations to decrease the side-channel leakage. While some other countermeasures such as masking involve modification of the source code which is labor-intensive and requires a thorough understanding of the crypto algorithm, our scheme is easy to design and requires no knowledge of the target algorithm. Our algorithms and tool can be applied to different cryptographic algorithms to help improve the system security.

V. Conclusion

In this paper, we present a method to automatically explore the design space for operation shuffling in cryptographic systems. Our method can detect the dependence among statements and guide automatic implementation of shuffling. We test the proposed scheme on Keccak. Results show that our algorithms and tool are efficient in transforming the source code for leakage reduction, effectively improving the resilience of cryptographic systems against power analysis attacks with less resource overhead than the secret sharing scheme. The future work will include combining the dependence analysis with leakage identification to aid efficient shuffling implementation at both the source code level and compilation stage.

Acknowledgment

This work was supported in part by National Science Foundation under grants SaTC-1314655 and MRI-1337854.

References

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology*, 1999, pp. 388–397.

[2] M.-L. Akkar and C. Giraud, "An implementation of DES and AES, secure against some attacks," in *Cryptographic Hardware and Embedded Systems*, 2001, pp. 309–318.

[3] M. Bucci, R. Luzzi, M. Guglielmo, and A. Trifiletti, "A countermeasure against differential power analysis based on random delay insertion," in *Circuits and Systems*, 2005, pp. 3547–3550.

[4] J.-S. Coron, "A new DPA countermeasure based on permutation tables," in *Security and Cryptography for Networks*, 2008, pp. 278–292.

[5] E. Prouff and R. McEvoy, "First-order side-channel attacks on the permutation tables countermeasure," in *Cryptographic Hardware and Embedded Systems*, 2009, pp. 81–96.

[6] N. Veyrat-Charvillon, M. Medwed, S. Kerckhof, and F.-X. Standaert, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *Advances in Cryptology ASIACRYPT*, 2012, pp. 740–757.

[7] A. G. Bayrak, F. Regazzoni, P. Brisk, F. X. Standaert, and P. Ienne, "A first step towards automatic application of power analysis countermeasures," in *Design Automation Conf.*, 2011, pp. 230–235.

[8] G. Agosta, A. Barenghi, and G. Pelosi, "A code morphing methodology to automate power analysis countermeasures," in *Design Automation Conf.*, 2012, pp. 77–82.

[9] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *Cryptographic Hardware and Embedded Systems*, 2012, pp. 58–75.

[10] A. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *Cryptographic Hardware and Embedded Systems*, 2013, pp. 293–310.

[11] H. Eldib, C. Wang, M. Taha, and P. Schaumont, "QMS: Evaluating the side-channel resistance of masked software from source code," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*, June 2014, pp. 1–6.

[12] R. A. Fisher, F. Yates *et al.*, *Statistical tables for biological, agricultural and medical research*. Edinburgh: Oliver and Boyd, 1963.

[13] "The danger of navet," http://blog.codinghorror.com/the-danger-of-naivete/.

[14] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[15] "Reference and optimized code in c," http://keccak.noekeon.org/KeccakReferenceAndOptimized-3.2.zip.

[16] D. Otte, "Avr–crypto–lib," *Online: http://www. das–labor. org/wiki/AVR–Crypto–Lib/en*, 2009.

[17] V. Paxson *et al.*, "Flex–fast lexical analyzer generator," *Lawrence Berkeley Laboratory*, 1995.

[18] "Evaluation environment for side-channel attacks," http://www.risec.aist.go.jp/project/sasebo/.

[19] M. Taha and P. Schaumont, "Differential power analysis of MAC-Keccak at any key-length," in *Advances in Information and Computer Security*, 2013, pp. 68–82.

[20] 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*, 2014.

[21] P. Luo, Y. Fei, X. Fang, A. Ding, D. Kaeli, and M. Leeser, "Side-channel analysis of mac-keccak hardware implementations," in *Hardware and Architectural Support for Security and Privacy*, 2015.

[22] 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*, 2014, pp. 147–169.

[23] J. Waddle and D. Wagner, "Towards efficient second-order power analysis," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, 2004, vol. 3156, pp. 1–15.

[24] P. Belgarric, S. Bhasin, N. Bruneau, J.-L. Danger, N. Debande, S. Guilley, A. Heuser, Z. Najm, and O. Rioul, "Time-frequency analysis for second-order attacks," in *Smart Card Research and Advanced Applications*, 2014, vol. 8419, pp. 108–122.

[25] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Building power analysis resistant implementations of Keccak," in *Second SHA-3 candidate conference*, 2010.