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

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

***Abstract*—Side-channel attacks have been a real threat against many critical embedded systems that rely on cryptographic algorithms as their security engine. 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 utilizing the independence among operations in cryptographic algorithms and randomizing their execution order. We design a tool to automatically detect such independence between 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 has effectively implemented 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. The most commonly used side-channel leakage is power consumption [1], in addition to other types like electromagnetic (EM) emanations [2] and timing information [3]. Various countermeasures have been presented to protect embedded systems from power analysis attacks. Secret sharing (random masking) [4] 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 against power analysis attacks, such as random delay [5] and shuffling [6], 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 [6], [7], random permutation is 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 [8], a simplified version of shuffling, Random Start Index (RSI), is presented which is easier to implement but can be significantly weaker than the random permutation.

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 [9], [10], [11], [12], [13] indicate a nascent trend towards automating the application of countermeasures to increase the security of the systems against power analysis attacks. They have focused on masking AES, including automatic instruction sensitivity quantification and local random precharging [9], a general code morphing engine design with alternative code segments that mitigate power leakage [10], compiler assisted masking implementation [11], and automatic security evaluation and verification [12], [13]. 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 is both algorithm and platform independent, and therefore can effectively reduce the power leakage and yet efficient to implement. We test our algorithm on Keccak, the new SHA3 standard, and the side-channel power analysis 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 any cryptographic algorithm.

The rest of this paper is organized as follows. Section II introduces some preliminaries on shuffling and basics of Keccak. In Section III, we present our algorithm to extract dependence relationships 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

In this section, we introduce the concept of shuffling, various data dependencies among operations and the rules of data consistency. As an example to show independent operations and shuffling, some basic details of Keccak are also introduced in this section.

A. Countermeasure to Power Analysis Attacks - Operation Shuffling

Operation shuffling is an effective countermeasure to mitigate the vulnerability of cryptographic systems against differential 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, for a large number of power leakage traces, the leakage of such intermediate variable is randomly spread onto n_p time points, and the corresponding correlation and signal-noise-ratio (SNR) for power analysis attacks will be effectively decreased to $1/n_p$ [6], [7].

Assume one piece of software code 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 $\{0, 1, \cdots n-1\}$. Algorithm 1 below illustrates the implementation of operation shuffling.

Algorithm 1 Shuffling execution

Input: $C = \{C_0, C_1, \cdots, C_{n-1}\}$, and a random order array, Order
1: **for** $i = 0 \rightarrow n-1$ **do**
2: **switch** Order$[i]$ **do**
3: **case** 0
4: Execute C_0
5: **case** 1
6: Execute C_1
7: $\cdots$
8: **case** $n-1$
9: Execute C_{n-1}
10: **end for**

Shuffling implementation includes two critical steps, extraction of statement dependency and generation of the random execution order array Order. Fisher-Yates algorithm [14], shown in Algorithm 2, is an efficient algorithm for generating such an array. It is an unbiased algorithm, which means the resulting Order array will be fully random [15]. Its computational complexity is linear, $O(n)$, where n is the array size. Thus we adopt this algorithm for random order array generation and expect it to impose small execution overhead in large crypto systems.

B. Data Dependencies and Operation Consistency

For operation shuffling, the most important step is to extract the dependency relationships of statements so as 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 $A = \{A[0], A[1], \cdots, A[N-1]\}$, in which N is the number of statements, also called the size of the code. 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 $V[i] = \{v_i[0], v_i[1], \cdots\}$. Each statement can have different number of variables, where $v_i[0]$ is statement $A[i]$'s only output and other variables are its inputs.
- For statements $A[i]$ and $A[j]$, with $i > j$, if there is data dependence between the two statements, we say that $A[i]$ depends on $A[j]$, denoted as $A[i] \Rightarrow A[j]$.

Algorithm 2 Permutation table generation with Fisher-Yates algorithm

Input: The size of the permutation table, n
Output: A permutation table Order at size n
1: Generate an array Order$= \{0, 1, \cdots, n-1\}$
2: **for** $i = n-1 \rightarrow 1$ **do**
3: $j \leftarrow$ random integer $0 \leq j \leq i$
4: Exchange Order$[j]$ and Order$[i]$
5: **end for**

There are three type of data dependencies between two sequential statements, $A[i]$ and $A[j]$, where $i > j$, defined in [16]:

- 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 dependency if no usage of the variable is found between the two statements. At the source code level, we keep such data dependency.

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

Example 2.1: For a piece of code:

$$\mathsf{a} = \mathsf{b{+}c{+}d}$$
$$\mathsf{e} = \mathsf{a{+}f}$$
$$\mathsf{a} = \mathsf{g{+}h}$$

They are denoted as a sequence of statements $A = \{A[0], A[1], A[2]\}$, where:

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

There exists RAW dependence between $A[1]$ and $A[0]$, WAR dependence between $A[2]$ and $A[1]$, and WAW depen-

dence 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 relationships.

C. Operations in Keccak

We choose Keccak [17], the newly selected new hash standard SHA-3 as an examples to apply the operation shuffling tool on. As the security of these primitives depends heavily on statistical properties of the operations and the key length, there is rich and iterative data processing and many operations are independent, e.g., the individual key byte processing of each round in bock ciphers like AES, which is the fundamental for operation shuffling.

Keccak is based on the Sponge construction with iterative permutations. Each permutation function f works on a state at a fixed length. For Keccak, MAC-Keccak is recommended by the Keccak designers for message authentication using a secret key [17]:

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

where the key K is concatenated with the message M as the input for the hash function.

The default Keccak mode is Keccak-1600, with the bit length at 1600. All of the 1600-bit states are organized in a 3-D array, $5 \times 5 \times 64$, as shown in Figure 1. 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 1.

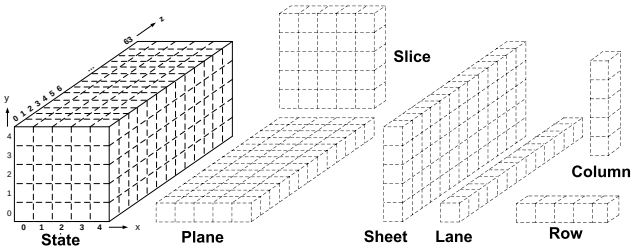

Fig. 1: Terminology used in Keccak

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

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

in which R_i stands for the input state of the i^{th} round, R_0 is the initial input determined by $K||M$, and the key bits occupying the bottom plane first. In this paper, we focus on the linear θ step for side-channel attacks. Details of other operations can be found in [17].

θ is a linear operation over 11 input bits and outputs a single bit, shown as follows:

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

The θ computation is done in two successive steps, θ_1 and θ_2. θ_1 first calculates the parity bit of each column, i.e., compressing a 1600-bit state into a 320-bit plane called the θ_{plane} by 320 independent bit operations:

$$\theta_{plane}(x, z) = \oplus_{y=0}^{4} S(x, y, z)$$
$$x \in \{0, 1, ..., 4\},\ z \in \{0, 1, ..., 63\}. \tag{4}$$

In the second step, θ_2 outputs the XOR between every bit of the state and two neighboring parity bits of the θ_{plane}:

$$\theta_{out}(x, y, z) = S(x, y, z) \oplus \theta_{plane}(x-1, z) \oplus \theta_{plane}(x+1, z-1)$$
$$x, y \in \{0, 1, ..., 4\},\ z \in \{0, 1, ..., 63\}. \tag{5}$$

The intermediate result of operation θ_1 in the first round, θ_{plane}, contains key information directly and has been used to extract the secret key in previous work [18], [19], [20], [21]. The θ_1 computation consists of many independent operations on key segments, i.e., it is leaky and suitable for operation shuffling. It is much harder to extract independent key-sensitive intermediate variables from later operations and rounds though. In this paper, we will use the leakages of θ_{plane} to discuss the effect of our algorithm.

III. Methodology and Algorithm

In this section, we describe our methodology of extracting the operation dependencies in cryptographic software, and the algorithm for automatic operation shuffling at the source code level.

A. Statement Dependencies Extraction

The simple Example 2.1 shows that the data dependence relationship 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 dependencies of these N statements, and it should only be lower-triangular. With the row index i and the column index j, if $M[i][j] = 1$, it means that $A[i] \Rightarrow A[j]$ for $0 \leq i, j \leq N-1$, $j < i$. The algorithm for dependence matrix generation is shown in Algorithm 3.

Algorithm 3 Dependence extraction

Input: The source code of the crypto algorithm
Output: Lower triangular dependence matrix M for the crypto 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 3, we can extract the dependence relationships of all the statements. This dependence matrix can help designers to automate shuffling. To find the statements group which allows shuffling is to find a sub-matrix in M along the main diagonal with all its elements zero, i.e., the corresponding statements do not depend on each other.

We use the Keccak source code (Keccak-simple32BI provided by [22]) as example here and apply Algorithm 3. There are 344 statements (SSAs) for the first two rounds (in one loop), and thus the dependence matrix size is 344×344. We only show the first 36×36 dependency sub-matrix for the first 36 statements in Fig. 2 due to the space constraint.

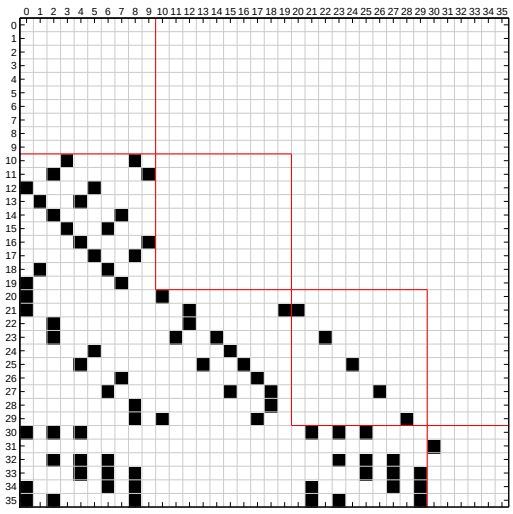

Fig. 2: The dependence relationship of the first 36 statements of Keccak

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 independent on each other, 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, the group of statements $\{A[21], A[23], A[25], A[27], A[29]\}$ 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 analyze the AES code from [23] using our algorithm. There are 55 statements for each round after expansion. Similar dependence result is obtained for AES, and more details will be given in Section III-B2. For larger cryptographic software, large N and complex dependencies will make the manual shuffling impossible. Automatic methods should be designed to find the shuffling space.

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 data independence. 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 do not depend on each other and their orders can be permuted. However, the space for shuffling at the basic block level may be limited. To fully explore the space for shuffling, 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 relationships among sensitive statements extracted, we can exploit it for automatic shuffling. We start from a simple case of single leakage, i.e., one sensitive statement for one intermediate variable, and then discuss the more general and complex cases of many leakages from many statements.

1) Random Insertion of One Statement: We assume that only one statement $A[m]$ in the code contains the sensitive information. Thus the problem reduces to insert $A[m]$ into the program code randomly with the number of possible position of $A[m]$ be maximum. We classify all the statements into three categories. For the data flow graphs of the program, all the statements that $A[m]$ directly and indirectly depends on should be executed before $A[m]$, and we put them into the category of Ancestors. All the statements that depend on $A[m]$ should be executed after $A[m]$, and we put them into the category of Offspring. The method to extract Ancestors based on the dependency relationship matrix is shown in Algorithm 4 and the method to extract Offspring is similar.

Algorithm 4 Random insertion of $A[m]$

Input: The $N \times N$ dependence matrix M, the index m of the statement containing the sensitive information
Output: list Ancestors

```
1:  Ancestors.insert(m), finished = 0
2:  while (!finished) do
3:      finished = 1
4:      for i=0 → Ancestors.size do
5:          for j=0 → Ancestors[i] do
6:              if (M[Ancestors[i]][j] = 1) & (j ∉ Ancestors) then
7:                  Ancestors.insert(j), finished = 0
8:              end if
9:          end for
10:     end for
11: end while
12: Ancestors.remove(m)
13: AscendSort(Ancestors)
```

After extracting these two lists, the rest of the statements are all assigned to a list Independent, which are all independent on $A[m]$. There may be data dependencies between statements in the list Independent and the original schedule in the program source code should be kept after we reschedule the program: all the statements in Ancestors are executed first and then Independent, and Offspring at last. For execution, only the statement $A[m]$ is floating and can be inserted to any place of Independent and leakage of $A[m]$ is spread from one

point to |Independent| points. The techniques above are based on the well-known code scheduling theory employed in the field of compiler optimization. For Keccak, assume the 50^{th} statement contains the secret information and designers want to insert it to other positions, using our algorithm, we find the Ancestors list with size 20, Offspring list with size 205, and Independent size 118. Which means that the 50^{th} statement can be inserted randomly at any one of 118 positions.

2) Shuffling of All Statements: Method shown in Section III-B1 is to explore the largest number of possible positions for one single statement which contains the secret information. This method will reduce the leakage of one key byte/word, but is not applicable to decrease all leakages of different key bytes. Next we devise another method for shuffling all leaky statements. We separate all the statements into different levels, such that the statements at the same level are independent of each other and can be permuted randomly without violating the rule of operation consistency. Statements at different levels have data dependencies and should preserve a certain execution order: statements at the lower level should be executed before statements at the higher level.

Algorithm 5 shows the procedure to separate all statements into different levels based on the dependence relationship matrix. It is actually running a reverse depth-first search on the data flow graphs of the program, and assign the statements (nodes in the DFGs) to different depths.

Algorithm 5 Execution level extraction

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

```
Initialize an zero array L[N]
for i = 1 → N − 1 do
    level = 0, newlevel = 0
    for j = 0 → i − 1 do
        if M[i][j] = 1 then
            newlevel = L[j] + 1
            if newlevel > level then
                level = newlevel
            end if
        end if
    end for
    L[i] = level
end for
```

We apply our algorithms on AES and Keccak, and the results show that our algorithms can effectively extract the dependence relationship 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 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. Statements at each level can be executed in a random order while the execution order between levels is specified. If the number of statements in one level is too small, the shuffling space is limited. To solve this problem, some dummy statements can be added into this level to increase the shuffling space but with 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 dependencies. Only a portion of them are leaky statements and permutation on them would decrease the side-channel leakage.

IV. Experimental Results and Evaluation

A. The Tool for Source Code Transformation

Our algorithms shown above operate on statements at the source code level without need to understand the target cryptographic algorithm. Therefore, tools based on the proposed algorithms only need some lexical analysis, but no complex semantic analysis like compilers do. In this work, we develop a source code transformation tool based on our algorithms using Python and its strong text processing packages. The structure of our tool is shown in Figure 3.

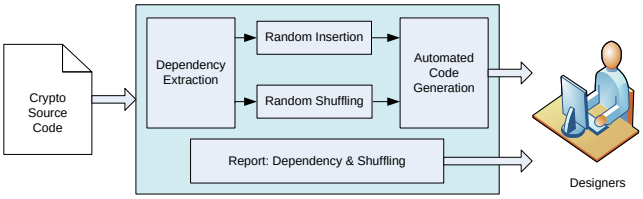

Fig. 3: Automatic source code transformation with shuffling

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 exists mature tools that can be applied, for example, we use FLEX [24] here.
2) Dependence analysis: Use Algorithm 3 shown in Section III to analyze the dependence relationships among the statements in source code and generate the dependence matrix.
3) Space exploration: Use Algorithms 4 and 5 to explore the space for random insertion and statement shuffling.
4) Code generation: Based on the shuffling space exploration results, embed an run-time permutation engine according to Algorithm 1 that contains code for generating the random execution order array given by Algorithm 2, and generate a report to the designer about dependence relationships 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 both effective in reducing the leakage and still efficient.

The designers only need to run this tool one time at the design stage. The generated code will be in the format shown in Algorithm 1 and contains code that generate the random order array at run-time. The computational complexity of Algorithm 3, 4 and 5 are all $O(n^2)$. For one round of Keccak which contains 344 statements, our Python implementation needs less than one second to finish all the work including dependency extraction and shuffling generation. The code will then be compiled before running on an experimental platform. Note that the transformations done by our tool will be orthogonal to normal compilation process and will not be changed. At runtime, an 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 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 MAC-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 [25].

We use similar setting for Keccak as previous papers [19]. Without loss of generality, we assume the key size is 320 bits, i.e., occupying the first plane. For software implementations, there are many intermediate variables that can be used for side-channel analysis. The same as previous works [19], we attack the first step of θ which compresses five planes (including the key plane) into one plane.With our dependency detection algorithm, statements of θ_1 are all separated into the first level which contains 10 statements. Meanwhile, the results of θ_1 are called 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 use both correlation power analysis (CPA) and mutual information analysis (MIA) [26] to attack the targeted implementations, which give similar results. Due to page limit we only present and discuss CPA results in this paper.

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. 4.

Fig. 4(a) shows the correlation result between the Hamming weight of eight bits of θ_1 output and power consumption 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 are big (0.55 and 0.27) and stand out from the noisy correlations at other non-leaky time points (varying between 0.1 and -0.1). For the shuffled Keccak implementation, the peak correlation is much smaller with the countermeasure and therefore it requires more power traces to reduce the variance of correlations at the other non-leakage points. With $15,000$ power traces the smaller peak leakage correlations are clearly visible in Fig. 4(b). Note the number of traces would not affect the peak correlations at the leaky time points, but just reduce the noisy correlations with more traces for better visualization. For shuffled implementation of Keccak, 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 correlation of the original implementation. The largest correlation is only about 0.09. With the decrease of correlation (SNR), the trace numbers needed for protected implementation will be 36 times that for unprotected implementation to achieve the same success rate by the formula in [27].

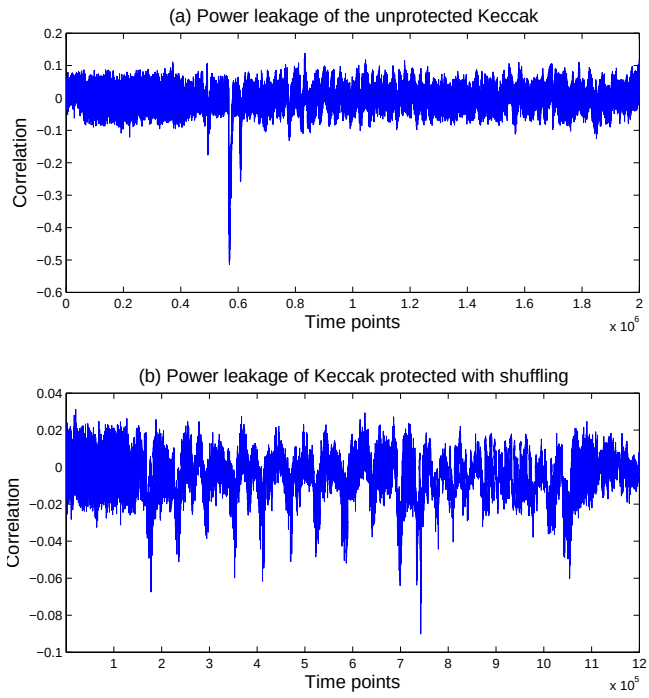

Fig. 4: Power leakage of the unprotected and shuffled Keccak on Microblaze

Note that Fig. 4 only shows the leakage reduction of one key byte. With group of statements permuted and 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 [28], [29]. 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 much smaller 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 that the secret sharing scheme is a manual two-share masking implementation and one random number is involved for masking [30]. The shuffling implementation is the one that our automation tool generates.

TABLE I: Resource and execution comparison of different schemes

Implementations	File size (Byte)	Clock cycles
Original [22]	31040	1670
Shuffling	40128	2580
Secret Sharing [30]	69272	6780

All these 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 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 the modification of source code which is labor-intensive and requires a thorough understanding of the crypto algorithm, our scheme is easy to design and it requires no knowledge of the target algorithm. Our algorithms and tool be applied to different cryptographic algorithms to help improve the system security.

V. Conclusion

In this paper, we present a method to explore the design space for operation shuffling in cryptographic systems automatically. 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 automatic leakage analysis to aid efficient implementation of source code statements shuffling and finer-grained compiler-assisted implementations.

References

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology CRYPTO 99*, 1999, vol. 1666, pp. 388–397.

[2] D. Agrawal, B. Archambeault, J. Rao, and P. Rohatgi, "The EM side–channel(s)," in *Cryptographic Hardware and Embedded Systems – CHES 2002*, 2003, vol. 2523, pp. 29–45.

[3] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[4] M.-L. Akkar and C. Giraud, "An implementation of DES and AES, secure against some attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2001*, 2001, vol. 2162, pp. 309–318.

[5] 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. ISCAS 2005. IEEE International Symposium on*, May 2005, pp. 3547–3550 Vol. 4.

[6] J.-S. Coron, "A new DPA countermeasure based on permutation tables," in *Security and Cryptography for Networks*, 2008, vol. 5229, pp. 278–292.

[7] E. Prouff and R. McEvoy, "First-order side-channel attacks on the permutation tables countermeasure," in *Cryptographic Hardware and Embedded Systems - CHES 2009*, 2009, vol. 5747, pp. 81–96.

[8] 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*, 2012, vol. 7658, pp. 740–757.

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

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

[11] A. Moss, E. Oswald, D. Page, and M. Tunstall, "Compiler assisted masking," in *Cryptographic Hardware and Embedded Systems - CHES 2012*, 2012, vol. 7428, pp. 58–75.

[12] A. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *Cryptographic Hardware and Embedded Systems - CHES 2013*, 2013, vol. 8086, pp. 293–310.

[13] 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.

[14] R. A. Fisher, F. Yates *et al.*, *Statistical tables for biological, agricultural and medical research*. Edinburgh: Oliver and Boyd, 1963.
[15] "The danger of navet," http://blog.codinghorror.com/the-danger-of-naivete/.
[16] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.
[17] G. Bertoni, J. Daemen, M. Peeters, and G. Assche, "The Keccak reference," *Submission to NIST (Round 3)*, January, 2011.
[18] M. Taha and P. Schaumont, "Side-channel analysis of MAC-Keccak," in *Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on*, June 2013, pp. 125–130.
[19] ——, "Differential power analysis of MAC-Keccak at any key-length," in *Advances in Information and Computer Security*, 2013, vol. 8231, 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 (ReConFig), 2014 International Conference on*, Dec 2014, pp. 1–7.
[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] "Reference and optimized code in c," http://keccak.noekeon.org/KeccakReferenceAndOptimized-3.2.zip.
[23] D. Otte, "Avr–crypto–lib," *Online: http://www. das–labor. org/wiki/AVR–Crypto–Lib/en*, 2009.
[24] V. Paxson *et al.*, "Flex–fast lexical analyzer generator," *Lawrence Berkeley Laboratory*, 1995.
[25] "Evaluation environment for side-channel attacks," http://www.risec.aist.go.jp/project/sasebo/.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Building power analysis resistant implementations of Keccak," in *Second SHA-3 candidate conference*. Citeseer, 2010.