VLSI Design Implementation of Advanced Encryption Standard Including Channel Attacks

Anand Mohan Thakur[1], Sachin Bandewar[3]
[1]Research Scholar 2Assistant Professor
[1,2] Department of ECE, Sri Satya Sai College of Engineering, RKDF University, Bhopal, M.P
[1]thakuranand143.amt@gmail.com, [2]sachin.bandewar9@gmail.com

***Abstract*--Cryptographic implementation is one of the vital applications for FPGAs with security as its major standpoint. But it still requires a lot of efforts to keep it aloof from attacks like Side Channel Attacks (SCA). One of the major attacks that threaten the security of FPGA implementation of cryptographic algorithm is Differential Power Analysis (DPA). In this paper, we have discussed various approaches to defend against DPAs. These new programs suggested in the different approaches with some results aid in evenly distributing the power consumption in FPGAs making it less prone**

Keywords--Side channel attack, differential power analysis, FPGA, randomized execution

I. INTRODUCTION

FPGAs have the major benefit in the proficiency of hardware design implementations. However, in the background of cryptographic implementations, the substantial defense, rather than the efficiency, is a more essential metric in the FPGA design. FPGAs are susceptible to some assault advancement such as side channel attacks. A major source of information in side channel attack is power consumption. The original message from cipher text and the key format can be assumed from the cryptographic procedures in encrypted FPGA implementation, by scrutinizing the correspondence involving the instantaneous power indulgence and transitional consequences relating the secret key. In the field of information security, the concern for security of FPGAs has been pivotal.

In 1996, Kocher [9] introduced the concept of SCA. In cryptographic implementation [10] DPA is considered as the most powerful attack out of all SCAs. By analyzing the correlation within the processed data and the power trace, the secret key of a cryptographic device is assumed in DPA statistically. The secret key is the most important means to explore the power consumption. An intruder first observes the initial m encryption operations, tracing the power consumption throughout the channel T1..M[1..k], k samples per trace, and cipher texts C1..m.

Assuming a secret keys Ks for verification, a selection function D(Ci, b,Ks) which computes the value of an in DPA. The difference between the average of the traces for which D(Ci, b,Ks) is 1 and the average of the traces for which D(Ci, b,Ks) is 0 is evaluated as the differential trace ΔD[1..k].

A term known as DPA peak[1] is explained where, when Ks is incorrect, the bit from the D function will differ from the actual target bit for about half of the cipher text Ci. Theoretically, when "1" bits inside the algorithm is uniformly distributed and the value of b as well as the text message is selected properly, the value of ΔD[b] is non- zero for a correct hypothesis Ks. For incorrect keys, ΔD[b] tends to 0 and no significant peak appears. If this DPA peak can be removed and the differential trace can be made even, then the accurate key will be concealed and secluded from the intruders.

Abundant techniques have been proposed in software as well as hardware level to be implemented against DPA. In methods described in [3, 7, 11] prevention of leakage by corresponding hardware implementation is described, since the scope of leak in information due to the physical implementation of cryptographic algorithm is high. Software methods in the random sequence of operations [4,5,12] and data masking [6] were proposed to protect against DPA. Randomization of execution operations over the various cryptographic algorithms set a good method against DPA, but the proposed methods have not succeeded in implementing it. Specific cryptographic algorithm, such as DES, AES or RSA can only be adapted to this. In this paper, we adapt the Data Flow Graph (DFG) to represent cryptographic algorithms, and propose five concrete algorithms which can be adapted to different cryptographic strategies. A data flow graph (DFG) is a graphical depiction of the "flow" of data through an information system, sculpting its procedures.

The major contributions of this paper include:

1).Adapting the DFG to represent cryptographic algorithms; 2). Proposing novel algorithms for randomized execution which can be adapted to various cryptographic schemes; and 3). Conducting experiments to evaluate our proposed methods. The rest of this paper is organized as follows: Section II uses an example to illustrate the main ideas to achieve our

II. MOTIVATIONAL

In this section, a motivational example is provided to show the randomized executions of tasks in a loop can protect against DPSs. In this example, a body of a loop program is represented as the DFG shown as in Figure 1. There are five task nodes in this loop. The data dependency between two task nodes is represented by an edge. We assume that executing each task consume a fixed amount of power, 50 for node A, 60 for node B, 30 for node C, 70 for node D, and 20 for node E, respectively. If a sequence of nodes in the DFG meets all data dependencies, this sequence is called a legal schedule. Obviously, there are various legal schedules

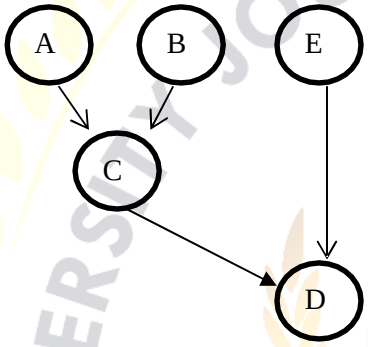

Figure 1: The DFG correspond with the loop in motivational example.

Conventionally, tasks in a loop body are executed with a fixed order in each iteration. In our example, assuming the loop is executed 5 times with the fixed order "A-B-E-D-C", the power consumption in each step is consistent, which equals to the power consumption of the corresponding task. After three iterations, the trace of the average power consumption of steps is "50-60-20-70-30". With this information, a hacker can easily get the exact order of the task execution by comparing values of this trace with the power consumption of each task.

However, when a loop is executed with different schedules in different iterations, the exact order of task execution is hard to observe. For instance, if the loop is executed 5 times with different schedules "A-B-E-D-C", "D-B-E-A- C", "A-E-B-C-D"," B-A-C-E-D", and "E-A-B-C-D", the

average power dissipations are represented in the Table 1, where the column "it" stands for iteration, "ES" stands for execution sequence, "S(i)" stands for step i. Hackers usually seize useful information by power analysis attacks. They analyzed distribution of the power values, and speculated the task node according to power values. So the bigger power gap between the two adjacent task nodes, the easier for hackers to attack. The more uniform the power dissipation distribution is, the more difficult it is to attack. The unbiased standard deviation of sequence "50, 60, 20, 70, 30" is 20.736, and the unbiased standard deviation of sequence "50, 48, 38, 40, 54" is 6.782. Obviously, the latter one has more uniform distribution, and it is uniform than that of the conventional fixed schedule. Thus, FPGAs can be protected against DPAs

Table 1: The power value sequence of random

It	ES	S(1)	S(2)	S(3)	S(4)	S(5)
1	A-B-E-D-C	50	60	20	70	30
2	D-B-E-A-C	70	60	20	50	30
3	A-E-B-C-D	50	20	60	30	70
4	B-A-C-E-D	60	50	30	20	70
5	E-A-B-C-D	20	50	60	30	70
Average Value		50	48	38	40	54

III. RANDOM

In this paper, we utilize a Data Flow Graph G =< V,E, p >, a weighted directed acyclic graph, where V is the set of task nodes, E ⊆ V ∗ V is the set of edges that define data dependencies, and p(v) is the power dissipation of a node v. We design randomized execution algorithms as shown in Figure 2. A controller works prior to each iteration. The DFG is the input for the controller, and the output of the controller is a task nodes execution sequence that is a

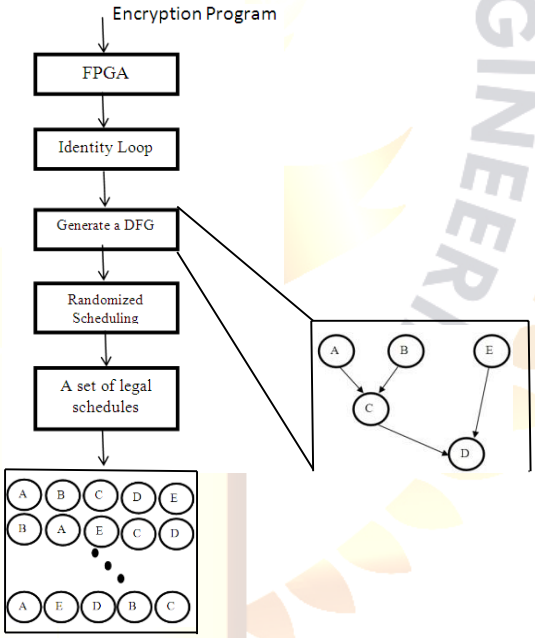

Figure 2: The flowchart of randomized execution

Algorithm 1: Randomized Ordering (RO)

Algorithm In the execution of a loop, certain task nodes are valid for executing in a given step. We define these task nodes as active task nodes of this step. The set of active task nodes is called the ready-to-run list. For randomized executing, we employ a random number generator (RNG) to choose a random

When a task node is chosen and executed, we remove it from the DFG, and then update the ready-to-run list. This process is repeated until all task nodes of the loop have been executed, as shown in Algorithm 3.1.
Generally, this algorithm will ensure randomized execution of a loop. But it will not defend against differential power analysis effectively in some special cases, such as a strictly sequential DFG. There is only one node ready in each step. Thus, the random execution is impossible in this case. To direct at which is an improved method is proposed in the next subsection.

Algorithm 3.1 Randomized Ordering (RO) Algorithm. Input:
Task Graph $G_t = \langle V,E \rangle$,
RNG. Output:
Every iteration of the loop is randomized for executing.
1: Schedule SL ← 0;
2: for iteration = 1 to n do
3: V ← Get task nodes set from original DFG;
4: $V_{leftover}$ ← V ;
5: for j = 1 to taskNum do
6: Ready-to-run List V_{RL} ← Get all active task nodes;
7: Array RL ← V_{RL};
8: $length_{RL}$ ← Get the length of Ready-to-run List; 9: rdm ← RandomNumberInIntervel[1, $length_{RL}$]; 10: SL ← SL ∪ RL[rdm];
11: $V_{leftover}$ ← $V_{leftover}$ − RL[rdm];
12: update the DFG after deleting the task RL[rdm];

Algorithm 2: Independent-Noise Randomized Ordering (INRO)

To solve the problem of random execution of a sequential DFG, a noise node concept is introduced. A noise node is
a special node of which the power dissipation is 0 or negligibly small. In a real world program, we can insert dummy instructions into the program to generate a noise node. After adding some noise nodes in the original DFG, we randomize the execution similar to the RO algorithm.
The INRO algorithm is capable in randomizing the execution of a strict sequential DFG. However, in some cases, a portion of the DFG is strict sequential, while the rest of the DFG can be executed in parallel. For example, in a DFG with a tree followed by a single path, noise nodes are more needed in the single path section, rather than in the tree section. Moreover, noise nodes in the tree section don't help in randomizing the execution, but hurt the execution speed of the system. The INRO algorithm faces issues in these cases, due to

Algorithm 3: Advanced Independent-Noise Randomized Ordering (AINRO) Algorithm

called Maximal Degree of Parallelism (MDP) for a DFG, the maximal number of task nodes, can be executed in parallel. In this algorithm, a redefined ready-to-run list is employed for optimizing the solution. The Ready-to-run list consists of active task nodes which are valid for executing in a given step, as well as noise nodes if necessary. Obviously the number of active task nodes is not more than the MDP of a DFG. To distribute more noise nodes the sequential part of the DFG and randomize the execution, we define three conditions for deciding the length of ready- to-run list;
1. It should be no larger than the MDP.
2. It should be no larger than twice the number of the active task nodes.
3. It should be no larger than the sum of number of active task nodes and the number of leftover noise nodes.
These three conditions for the length of ready-to-run list help in distributing noise nodes effectively. In each step, we choose a node randomly until a task node is chosen. Then we update the ready-to-run list and loop proceeds by this rule.
Algorithm 3.2 Advanced Independent-Noise Randomized Ordering (AINRO) Algorithm.
Input:
Task Graph $G_t = \langle V,E \rangle$, RNG, noiseNum.
Output:
Every circle of the loop is randomized for executing. 1: Schedule SL ← 0;
2: MDP ← Get the number of maximal parallel maximal task nodes;
3: for iteration = 1 to n do
4: V ← Get the Tasks Set from original DFG; 5: $V_{leftover}$ ← V ;
6: actNoise ← noiseNum; 7: for j ← 1 to taskNum do
8: AT ← Get a set of Active Task nodes from DFG ($V_{leftover}$);
9: Ready-to-run List V_{RL} ← AT 10: Array RL ← V_{RL};
11: $length_{AT}$ ← Get the length of AT; 12: $length_{RL}$ ← Min($length_{AT}$ + actNoise, 2 ∗ $length_{AT}$,MDP);
13: if $length_{AT}$ < $length_{RL}$ then
14: add ($length_{RL}$ − $length_{AT}$) noise nodes to RL; 15: end if
16: for i = 1 to $length_{RL}$ do 17: noiseNumCT ← 0;
18: rdm ← RandomNumberInIntervel[1, $length_{RL}$]; 19: SL ← SL ∪ RL[rdm];
20: if RL[rdm] is a noise node then 21: noiseNumCT + +;
22: RL ← RL − RL[rdm];
23: else
24: exeT ask ← RL[rdm]; 25: break;
26: end if
27: end for
28: $V_{leftover}$ ← $V_{leftover}$ − exeT ask;
29: actNoise ← actNoise −

Algorithm 4: Trapezoid Randomized Ordering (AINRO) Algorithm

In this subsection, a new conception is defined called Trapezoidal Decomposition for a DFG, which is we call two straight line segments in the plane noncrossing iff their intersection is either empty or a common endpoint. Consider a set S of n nonhorizontal, noncrossing closed line segments. Starting at each endpoint of each segment in S draw two horizontal rays, one towards the left and one towards the right, each extending until it hits a segment of S. For a segment endpoint p we call the union of these two possibly truncated rays emanating from ***p*** the horizontal extension through ***p.*** The segments of S together with the horizontal extensions through the endpoints form a plane graph, which we call the trapezoidation of S, or T(S) for short. As each face of T(S) has two horizontal sides (one of which might have length 0) we are justified in calling the faces of T(S) trapezoids.

Algorithm 3.4: Trapezoid Randomized Ordering (AINRO) Algorithm:: Before presenting the final algorithm and its analysis a bit of notation: Let $Log^{(i)}$ ***n*** denote the ith iterated logarithm, i.e. $log^{(o)}$ ***n*** = ***n*** and for i > 0 we have $log^{(i)}$ ***n*** = $log(log^{(i-1)}$ ***n).*** For ***n*** > ***0*** let log* ***n*** denote the largest integer 1 so that $log^{(l)}$ ***n*** ≥ 1, and for ***n*** > ***0*** and 0~ ***h s*** log* ***n*** let ***N(h)*** be shorthand for [*n* /$log^{(h)}$) *n]*.

The input to the algorithm below is a simple polygonal chain C of ***n*** segments in consecutive order along C.

1: Generate s_1, s_2, . . . s_n, a random ordering of the segments of C

2:Generate YI, the trapezoidation for the set {s,} along with the corresponding search structure

3:For *h* = 1 to log* *n* do

1) For *N(h* - 1) < i 6 *N(h)* do

2)Obtain trapezoidation z and search structure Z_i from Z_{i-1} and by inserting segment si.

3)Trace C through TN(k) to determine for each

Algorithm 5: Multilevel Randomized Ordering (MRO) Algorithm

In this subsection, a new concept for graph partitioning in an efficient way to minimize the sum of the weights of edge crossing between sets of a DFG. The algorithm consists of two loops nested. The outer loop allows attempted sequences until no further improvement of partitioning is necessary. The inner loop presides over a sequence of moves of vertices

Algorithm3.5: Multilevel Randomized Ordering (TRO) Algorithm

1:Until No better partition is
2: Best Partition: = Current
3: Partition Compute all initial
4: gains
5: Until Termination criteria
6: reached Select vertex to move
7: Perform move
8: Update gains of all neighbors of moved vertex
Partition Then Best Partition: = Current
9: End Until
10: Current Partition:= Best
Partition 11:End Until

IV. EXPERIMENTAL

In this section, we conduct experiments with our algorithms on benchmarks from Mibench [8]. The Sim-Wattch [2] is used as the power test platform. Various benchmarks are selected from Mibench and compiled as ARM-elf executable using a GNU ARM-elf cross compiler. The experiments are conducted based on the power model of the Intel(R) Core™2 Duo T9600 processor, which is of 2 cores, 2.8 GHz clock speed, 6M cache and 1066 MHz front side bus (FSB) speed. The memory of the system is 3002 MB DDR RAMs. We choose the Logistic Mapping as the random number generator.

In order to reduce the system overhead, the number of noise node should be limited. In our experiments, $\lfloor \frac{N_{node}}{10} \rfloor$ noise nodes are added to the original DFG in INRO algorithm and AINRO algorithm, where N_{node} represents the number of task nodes in original DFG.

Table II Resource

ALG.	FPGA	LUT	FF	Slices
AES	SPARTAN3E xc3s500e-fg320	4253 45%	453 4%	2226 47%
AES	VIRTEX2P Xq2vp70-5ff1704	4284 6%	451 0%	2235 6%
AES	VIRTEX 5 XUPV5-	11524 16%	7873	3312

Table III Resource

ALG.	Bond	GClK	Freq (MHz)	Area
AES	23 9%	2 8%	41.5	-
AES	23 2%	2 12%	43.5	-
AES	386	776K Gate		

The proof-of-concept has been prototyped using various FPGAs. Table II shows the logic and memory utilization of the various cryptographic algorithm in different FPGAs. The resource allocation is less than expected, and can be compared with their prior implementations in other works. Hence if implemented by the cloud providers can save a lot in infrastructure. Table III also shows the area and frequency requirement of the individual algorithms when implemented in various FPGAs.

When the DMP(Diverge Merge Processor) [15] of the object DFG is larger than the number of available independent noise nodes, the AINRO algorithm works as similarly as the INRO does. Thus, in such circumstances, the INRO algorithm is better due to the simpler program and the lower overhead on the power as well as the execution time. However, when a object

approximately sequential, AINRO algorithm has better performance. A performance evaluation of the above three algorithms can be verified in the research work

V. CONCLUSION

In this paper, we analyze the problem of randomized execution in a loop in order to protect FPGAs from DPAs. We propose few algorithms to randomize the execution in a loop. Algorithm RO, INRO and AINRO apply to the situation where all task nodes in a DFG cost the same amount of time in the execution. Algorithm TRO and MRO still are in a naïve stage, yet are effective in their performance. Experimental results show that the unbiased variance of a power sequence in a loop is decreased by 65.06% to 77.56%. Thus makes it difficult for attackers to analyze power effectively.

REFERENCES

[1] E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In Cryptographic Hardware and Embedded Systems, pages 16–29, 2004.

[2] J. Chen, M. Dubois, and P. Stenstrom. Simwattch: Integrating complete-system and user-level performance and power simulators. Micro, IEEE, 27(4):34–48, 2007.

[3] C. Clavier, J.-S. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In Cryptographic Hardware and Embedded Systems, pages 252–263, 2000.

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

[5] J. Daemen and V. Rijmen. Resistance against implementation attacks: A comparative study of the AES proposals. In Second Advanced Encryption Standard Candidate Conference, pages 122– 132, 1999.

[6] L. Goubin and J. Patarin. DES and differential power analysis the "duplication" method. In Cryptographic Hardware and Embedded Systems, pages 158–172, 1999.

[7] S. Guilley, L. Sauvage, F. Flament, V.-N. Vong,P. Hoogvorst, and R. Pacalet. Evaluation of power constant dual-rail logics countermeasures against DPA with design time security metrics. IEEE Transactions on Computers, 59(9):1250–1263, September 2010.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M.Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In IEEE International Workshop on Workload Characterization, 2001. WWC-4, pages 3– 14, 2001.

[9] P. C. Kocher, J. M. Jaffe, and B. C. Jun. Differential power analysis. In Advance in Cryptography - CRYPTO 1999, pages 388–397, 1999.

[10] S. Mangard, E. Oswald, and T. Popp. Power analysis attacks: revealing the secrets of smart cards. Springer-Verlag, 2007.

[11] A. Moradi, M. Taghi, M. Shalmani, and M. Salmasizadeh. Dual-rail transition logic: A logic style for counteracting power analysis attacks. Computers and Electrical Engineering, 35(2):359– 369, March 2009.

[12] J. S. Pan, B. L. Guo, and A. Abraham. Resistance DPA of RSA on smartcard. In International Conference on Information Assurance and Security, pages 406–409, 2009.

[13] M. Qiu and E. H.-M. Sha. Cost minimization while satisfying hard/soft timing constraints for heterogeneous embedded systems. ACM Transactions on Design Automation of Electronic Systems (TODAES), 14(2):Article 25, 1–30, Mar. 2009 (TODAES

[14] Jiayin Li, Daidu Zhang, M. Qiu, J Shen. Security Protection on FPGA against Differential Power Analysis Attacks. In CSIIRW '11, October12-14, Oak Rigde, USA ACM 978-1-4503-0945-5 ISBN

[15] Hyesoon Kim Jos´e A. Joao Onur Mutlu, Yale N. Patt. Diverge-Merge Processor (DMP): Dynamic Predicated Execution of Complex Control-Flow Graphs Based on Frequently Executed Paths, The 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)0-