ERIST: An Efficient Randomized Instruction Insertion Technique to Counter Side-Channel Attacks

Zhangqing He, Tianyong Ao, Meilin Wan, Kui Dai, and Xuecheng Zou, *Member, IAENG*

***Abstract*—This paper proposes an Efficient Randomized Instruction inSertion Technique (ERIST) to resist side-channel attacks (SCAs). In ERIST, an instruction insertion hardware module is embedded into a processor that generates random real instructions and inserts them into the execution stream of cryptographic programs. ERIST scrambles the power profile of a cryptographic application and can resist the latest signal processing and modeling attacks. An instruction insertion strategy is adopted to control the generation and insertion of random instructions, which greatly improves the efficiency of ERIST. Theoretical analysis and simulated correlation power analysis (CPA) attack results show that this technique is significantly more secure and more efficient than previous similar countermeasures.**

***Index Terms*—side-channel attacks, DPA, countermeasures, randomized instruction insertion**

I. INTRODUCTION

Side-channel attacks (SCAs) [1],[2] pose a serious threat to the security of cryptographic modules. Their targets range from primitives, protocols, modules, and devices to entire systems. Protecting cryptographic implementations against SCAs is a challenging task.

Random Delay Insertion (RDI) [3],[4] is a simple but effective countermeasure against SCAs and fault attacks. In most side-channel and fault attacks, adversaries are required to know the precise time when the target operations occur in the execution flow of a cryptographic algorithm. Introducing random delays into the execution flow breaks synchronization and increases the complexity of the attack. The RDI countermeasure has no dependency on algorithms, and can be applied to different implementation platforms, i.e. microprocessor, ASIC and FPGA. Furthermore, a combination of RDI and other countermeasures, such as masking, are often used in real-life protected implementations to resist powerful SCAs [5].

Random delays can be introduced in software or hardware. The insertion of dummy operations is a popular software method [6],[7]. However, dummy instructions have their own power profile, and the latest research shows that inserted dummy instructions can be removed from the instruction sequence by appropriate signal processing or modeling tools, e.g., cross-correlation or hidden Markov models (HMMs) [8]. HMM is a probabilistic model generally used for data classification [9]; Durvaux et al. successfully used it to remove the random delays generated by software methods [10].

Random delays can also be introduced in hardware. Bucci et al. [11] proposed architecture for delaying generation at the gate level, and Lu et al. [12] implemented it on an FPGA and addressed the optimization of delay generation parameters for this architecture. However, these gate-level delay insertion schemes bring high area and run-time overheads [13]. Non-deterministic processors [14],[15] can randomly change the sequence of some independent operations and scramble the power profiles of cryptographic programs, but the number of such operations in cryptographic algorithms is usually limited, thus reducing the practicability of these techniques. Ambrose et al. [16] presented a randomized instruction injection technique (RIJID), which inserts some real instructions into the protected instruction flows. RIJID overcomes some weaknesses of the dummy insertion techniques to some extent. However, RIJID does not consider optimization of the delay distribution and leads to a lack of efficiency.

In this study, we propose an efficient RDI technique to counter side-channel attacks. We make two contributions by proposing the following:

1. A new framework of randomized instruction insertion technique called Efficient Randomized Instruction inSertion Technique (ERIST) that can randomly insert random real instructions into an execution flow, thus scrambling the power profiles of the cryptographic program. ERIST can resist the latest signal processing and modeling attacks.

2. A method for generating and inserting random instructions that improves the statistical distribution of random delays and greatly increases the efficiency of ERIST.

We implemented ERIST on an ARM7 core and mounted simulated CPA attacks. The results show that ERIST is significantly more secure and efficient than published similar solutions.

Manuscript received August 18, 2015; revised November 01, 2015. This work is supported by the National Natural Science Foundation of China (No.61376031) and Science and technology project of Henan Provance(152102210055).

Zhangqing He is with the School of Optical and Electronic Information, Huazhong University of Science and Technology, Wuhan, China and is also with the Hubei Collaborative Innovation Center for High-efficiency Utilization of Solar Energy, Hubei University of Technology, Wuhan, China(corresponding author, phone: +86-02759750879; e-mail: ivan_hee@126.com).

Tianyong Ao is with the School of Physics and Electronics, Henan University, Kaifeng, China and is also with the School of Optical and Electronic Information, Huazhong University of Science and Technology, Wuhan, China

Meilin Wan, Kui Dai, and Xuecheng Zou, are with the School of Optical and Electronic Information, Huazhong University of Science and Technology, Wuhan, China, and also with Innovation Center for MicroNanoelectronics and Integrated System, China.

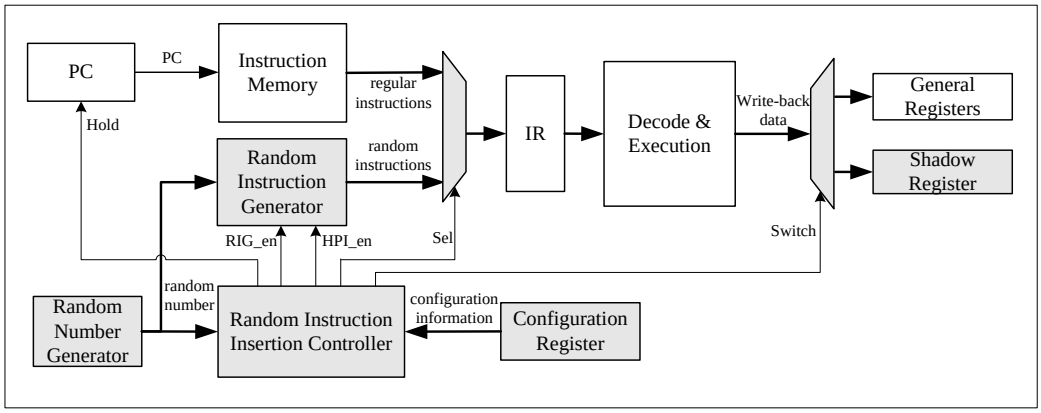

Fig. 1. ERIST framework.

II. ERIST FRAMEWORK

A. Overview of ERIST Framework

The basic idea of ERIST for countering SCAs is to embed a random instruction generation and insertion module into a processor. The module can generate and insert some random real instructions, such as AND and XOR, into the instruction stream when the processor executes the protected cryptographic codes. The injected instructions are random and actually executed, thus it is impossible to identify and remove them from their power sequences. Several special strategies ensure that the execution of random instructions does not change the execution results of protected codes.

The ERIST framework is shown in Figure 1. Five hardware components (denoted as gray modules) are added into the original core of a processor. We introduce the details of these components in the following subsections.

B. Random Instruction Generator

A random instruction generator produces random instructions used for injection into the processor. In order to confuse the adversary and reduce hardware overhead, the type of instructions that can be generated should be carefully selected. Given that the main SCA targets are symmetric and asymmetric encryption algorithms, the major operations in the symmetric encryption algorithms are logic and arithmetic instructions, such as XOR and AND. In order to protect the symmetric encryption algorithms, inserting some random logic and arithmetic instructions would be sufficient. However, the asymmetric encryption algorithms usually contain many high-power instructions (HPIs), such as "multiply." The power consumption of this type of instruction is significantly larger than that of the logic and arithmetic instructions. Hence, some random HPIs should be inserted in order to protect the asymmetric encryption algorithm.

Thus, our random instruction generator can produce two types of instructions, logic and arithmetic, in addition to some HPIs. The type of HPIs that can be generated depends on the instruction set of the processor on which ERIST is implemented. The instruction insertion controller that uses an *HPI_en* signal determines whether to generate HPIs.

C. Random Instruction Insertion Controller

The random instruction insertion controller is the core component of ERIST, and it controls the operation of other components. If the random instruction insertion function is triggered, the controller selects some specific time to insert some random instructions into the instruction stream according to an efficient instruction insertion algorithm (details of the instruction insertion algorithm are presented in Section III).

To start an injection, the controller generates five signals (*RIG_en*, *HPI_en*, *Hold*, *Sel*, and *Switch*) simultaneously to control the other modules (see Figure 1). The *RIG_en* signal informs the random instruction generator to produce appropriate random instructions. The *Sel* signal selects the generated random instructions and sends them to the instruction registers (IRs). At the same time, the *Hold* signal is generated to hold the program counter (PC).

D. Shadow Register

Implementation of the inserted random instructions actually affects the execution results of the protected encryption codes. This is because execution of these instructions rewrites the values of their destination registers, which can be used by subsequent regular instructions. To address this problem, we add a shadow register into the processor to work as the destination register of all the random instructions. When a random instruction is executed, the write-back data is written into the shadow register (informed by the *Switch* signal) regardless of the original destination register. This way, execution of all the random instructions does not amend the values of the general registers.

E. Configuration Register

To improve efficiency, a random instruction insertion configuration register (RIICR) is introduced in ERIST to configure its operating parameters.

There are three subsegments in RIICR: the enable bit (*En*), *HPI_en* bit, and setting bits for security levels (*SLs*). A typical format for RIICR is shown in Figure 2. If *En* is set to "1," the random instruction insertion function is triggered. The *HPI_en* bit determines whether to generate HPIs. The *SL* bits can determine the SCA resistance of the ERIST processor by changing the insertion frequency of random instructions (details are discussed in Section III).

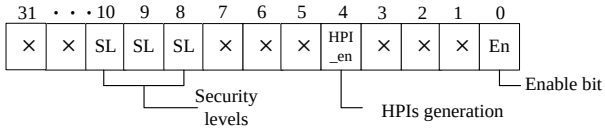

Fig.2 The arrangement of bits in RIICR

F. Software Implementation Flow

When a cryptographic application is executed on the ERIST processor, programmers select the critical code segments that need protection, and insert two write RIICR instructions at the start and end of the segments, respectively. The first write RIICR instruction sets the *En* bit in RIICR to trigger the generation and insertion of random instructions, and simultaneously sets the values of *HPI_en* and *SL* bits. The second write RIICR instruction clears *En* in order to stop generating and inserting random instructions.

Compared with adding special extended instructions in RIJID [16], our method does not need compiler support, which greatly increases availability. Furthermore, the operating parameters of ERIST can be configured as required. Thus, ERIST can be flexible and efficient in order to meet different security requirements.

III. Efficient Instruction Insertion Strategy to Improve Efficiency

A. Efficiency of Random Delay Insertion

Random delay techniques protect the cryptographic algorithms against side-channel attacks by introducing some delays into the cryptographic execution flows. A single delay can be easily removed by static alignment of side-channel traces. Therefore, the execution should be interleaved with delays in multiple places, so that an attacker is likely to deal with the cumulative delay of several random delays.

Recent studies show that the statistical distribution of cumulative delays greatly affects the resistance of random delay techniques against SCA [17]. The complexity of a DPA attack grows quadratically or linearly with the standard deviation of the trace displacement in the attacked point [7]. In order to improve efficiency, we should adhere to the following criteria for random delay generation.

1. The sum of random delays from start or end to some attack point within the execution should have the greatest possible variance.

2. The performance overhead should be as small as possible. That is to say, the mean of random delays should be as small as possible.

Most methods proposed for random delay generation usually insert multiple independent and identically distributed random delays in the encryption execution flows, e.g., RIJID [16]. According to the Central Limit Theorem, the distribution of the sum of N independent delays converges to normal with mean $N\mu_d$ and variance $N\sigma^2{}_d$, where μ_d and $\sigma^2{}_d$ are, respectively, the mean and variance of the duration of an individual random delay. In order to improve efficiency, we should modify the delay generation method to increase variation with the same mean in the cumulative distribution.

B. Proposed Random Instruction Insertion Method

In ERIST, introducing random delays depends on inserting random instructions. Therefore, we propose an efficient method for controlling the generation and insertion of random instructions and obtain an appropriate statistical distribution of cumulative random delays, which is approximated to the uniform distribution.

The following steps describe this method:

1. Initially, three non-negative integer parameters a, b, and k are chosen (a < b). Once these parameters are set, they do not change in an implementation of the algorithm.

2. If the enable bit of RIICR is set to 1, the processor starts generating and inserting random instructions. To do this, an integer value m' is first randomly and uniformly generated in the interval $[0,(a-b)\cdot 2^k)$.

3. At each interval of a fixed d regular instruction (d is obtained from RIICR), during execution of the protected algorithm, ERIST continuously generates and inserts c random instructions, where c is obtained by first generating a random integer $c' \in [m', m'+(b+1)\cdot 2^k)$ and then letting $c \leftarrow \lfloor c' \cdot 2^{-k} \rfloor$.

4. Once the enable bit of RIICR is set to 0, the processor stops generating and inserting random instructions.

C. Analysis

According to our method, if the number of instructions from start or end to the attacked point in a protected encryption code is l, the adversary considers N insertions in every execution, where $N = Int(l/d)$, and *Int(x)* denotes the largest integer that is lower than or equal to x. Figure 3 depicts two possible instruction sequences when $d = 4$, where c_i represents the number of inserted instructions in the *i-th* insertion of an execution, "×" denotes an injected random instruction, and "A~Z" denote regular instructions.

In order to simplify the analysis, we assume that each inserted random instruction has the same execution time, and one random instruction means one delay. It should be noted that if HPIs are inserted, the time delays caused by the same number of inserted instructions is different. However, this difference can be ignored because it has no obvious influence on method performance.

Now we provide an example to make the algorithm easier to understand. If we set the parameters $a = 5$, $b = 3$, and $k = 3$ at the beginning of each execution, m' is randomly and uniformly generated within the interval [0, 15]. During an execution, c' is generated randomly in $[m', m'+31]$ and $c = \lfloor c' \cdot 2^{-k} \rfloor$. Table I depicts the variation tendency of the probability density function of c with a different m'. We can see that when m' increases from 0 to 15, the mathematical expectation of c, denoted as $E(c)$, increases accordingly from 1.5 to 3.375.

ABCD×××ACDB××AADC×DHEF××××AIDG××ACBD……AFDG×××AAD
c_1=3 c_2=2 c_3=1 c_4=4 c_5=2 …… c_N=3

ABCD××ACDB××××AADC×DHEF××AIDG×××ACBD……AFDG××AADE
c_1=2 c_2=4 c_3=1 c_4=2 c_5=3 …… c_N=2

Fig. 3. Possible instruction sequence when $d = 4$.

TABLE I
PROBABILITY DENSITY FUNCTION OF c WITH DIFFERENT m' IN OUR INSTRUCTION INSERTION METHOD, WHERE $a = 5$, $b = 3$, $k = 3$

m'	Pr(c)						
	c=0	c=1	c=2	c=3	c=4	c=5	E(c)
m'=0	8/32	8/32	8/32	8/32	0	0	1.500
m'=1	7/32	8/32	8/32	8/32	1/32	0	1.625
m'=2	6/32	8/32	8/32	8/32	2/32	0	1.750
.				.			
.				.			
.				.			
m'=14	0	2/32	8/32	8/32	8/32	6/32	3.250
m'=15	0	1/32	8/32	8/32	8/32	7/32	3.375

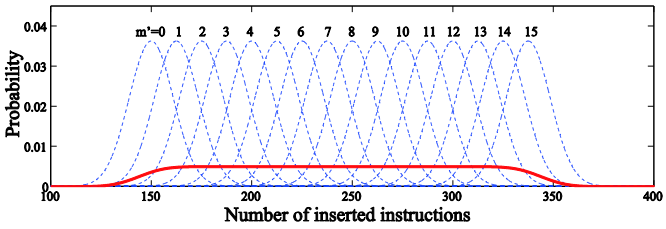

Fig. 4. Distribution of time delays caused by 100 cumulative insertions when $a = 5$, $b = 3$, $k = 3$.

Now we consider N insertions for each execution. The distribution of time delays caused by N cumulative insertions is approximated to a discrete normal distribution for a fixed m' (see the dotted lines in Figure 4). In an SCA, an adversary usually performs an algorithm many times in order to conduct statistical analysis. Because the value of m' varies from execution to execution and is produced randomly and uniformly, the entire distribution curve of time delays is an approximately uniform distribution (the heavy line in Figure 4).

D. Comparing with Plain Uniform Delays

For better comparison, we consider a typical method for delay generation: plain uniform delays. In this method, the number of inserted instructions in each insertion is uniformly distributed in the interval [0, a], the distribution of the sum of N insertions for the plain uniform delays converges to normal with mean $Na/2$, and variance $Var(S'_N) = N((a+1)^2 - 1)/12$.

For our method, the number of random instructions in an individual insertion is c and $c = \lfloor c' \cdot 2^{-k} \rfloor$, where c' is uniformly distributed in $[m', m' + (b+1) \cdot 2^k)$. We can consider c a random variable: $c = m + v_i$, where $m = \lfloor m' \cdot 2^{-k} \rfloor$ and v_i is a random variable in the interval [0, b + 1].

We have

$$E(S_N) = E(Nm) + E(\sum_{i=1}^{N} v_i)$$

$$= N \cdot \frac{a-b-1}{2} + N \cdot \frac{b+1}{2} = \frac{Na}{2} \qquad (1)$$

and

$$Var(S_N) \approx N^2 \cdot \frac{(a-b)^2 - 1}{12} \qquad (2)$$

Thus, the plain uniform delay has a variance of the sum of N insertions in $\theta(N)$, and our method has a variance in $\theta(N^2)$, as indicated in Table II. With the same mean $\frac{Na}{2}$,

TABLE II
DISTRIBUTION PARAMETERS OF TWO METHODS

	mean	variance
Plain uniform delay	$\frac{Na}{2}$	$\frac{N((a+1)^2 - 1)}{12}$
Our algorithm	$\frac{Na}{2}$	$N^2 \cdot \frac{(a-b)^2 - 1}{12}$

the ratio of $Var(S_N)$ to $Var(S'_N)$ is approximately equal to $N(\frac{1 - b/a}{1 + 1/a})^2$. Therefore, when N is large, our method is significantly more efficient than the plain uniform method.

E. Implementation of our Instruction Insertion Method

To implement our method, the key point is to calculate the number of inserted random instructions c for each insertion. To do this, we first need to produce the random numbers m' and c'. If the parameters are determined such that $(a-b) \cdot 2^k = 2^u$ and $(b+1) \cdot 2^k = 2^v$ (u *and* v are positive integers), m' and c' can be efficiently generated by a u-bit and v-bit random number generator. Then we obtain the value of c by abandoning the lower k bits of c'. Therefore, the cost of implementing our algorithm is very small.

F. Choosing Algorithm Parameters

According to Eqs. (1) and (2), if we select appropriate fixed values for a and b, the mean and variance of the cumulative delays are consequently determined by parameter N. Because $N = Int(l/d)$, parameter N is determined by d, which is inversely proportional to N for a fixed l. According to the theory of Tunstall and Benoit [7], a larger mean of cumulative delays represents greater performance overhead, and a greater delay variance indicates stronger resistance against SCA. Therefore, the mean of the delays should be as minimal as possible, thereby suggesting that a larger d should be used. However, as d increases, delay variance decreases, thus introducing a trade-off between overhead and system security. Hence, we should choose a different d for different security requirements in order to obtain the best trade-off between overhead and SCA resistance. In ERIST, d can be set by the setting bits of *SLs* in RIICR, as described in Section II.

IV. EXPERIMENT RESULTS

A. Implementation of ERIST on ARM7

We implemented our ERIST framework based on ARM7 processor core (ARM7TDMI-S), and called such implementation ERIST-ARM7.

In ERIST-ARM7, the random instruction generator can produce two types of instructions: single cycle data processing, and MUL and MLA. The data-processing instructions are generated according to the encoding diagram shown in Figure 5. The highest-order bit (bit 24) of op-code is set to 0 and the rest bits (bits 21 to 23) are filled with random numbers to ensure only single cycle data-processing instructions, such as AND, ADD, and XOR, are generated. Moreover, if the *HPI_en* bit in RIICR is set to 1, the MUL and MLA instructions are generated according to the

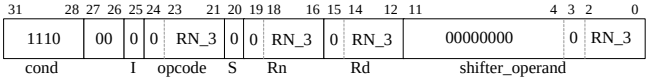

Fig. 5. Single cycle data-processing random instruction construct, RN_i, represents an *i*-bit random number.

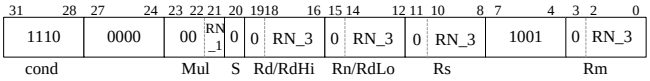

Fig. 6. Construct of multiply and multiply-accumulate random instructions.

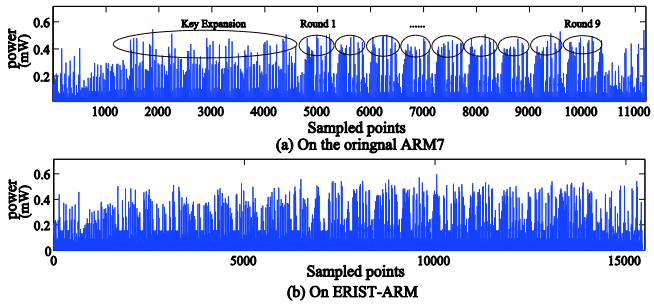

Fig. 7. Power simulation sequences for AES implementation on different platforms.

encoding diagram shown in Figure 6. All the random numbers used in ERIST-ARM7 are produced by a simple Linear Feedback Shift Register (LFSR) called a pseudo-random number generator (PRNG).

The other modules of ERIST-ARM7 are designed according to the methods described in Sections II and III. We implemented an ERIST-ARM7 core in Verilog and synthesized it through the Synopsys Design Compiler based on a UMC 0.18 μm standard cell process library. The results show that 32,494 GEs are required. This is an increase of only 2,413 additional GEs compared with the original ARM core area, which needs approximately 30,081 GEs.

Then we performed a standard AES algorithm on our ERIST-ARM7 core. The benchmark code and ERIST-ARM7 core were simulated together using the ModelSim HDL simulator, which generates a stimulus wave with switching information. The power values were measured with PrimePower, which provides measurements in watts (W). Figure 7 shows the power simulation sequences of the AES implementations. We can see that the AES encryption rounds can be identified easily when the AES codes are executed on an original ARM7 core. Once it is executed on the ERIST-ARM7 core, the process is disrupted and confused by inserted random instructions that cannot be identified.

B. Counteracting Against Cross-correlation and HMM Attacks

For cross-correlation attacks, in order to identify and remove a random instruction, the adversary should extract the pattern of an inserted instruction, and then match this pattern to clock cycles in the side-channel traces by computing the cross-correlation between them. However, in ERIST, the inserted instruction type is the same as that for the regular instructions, and thus it is impossible to extract a recognizable pattern for the inserted instructions.

For HMM attacks, adversaries first have to build a Markov model for the protected AES code, and then estimate the emission probability functions $e_i(l_t) = \Pr(l_t \mid s_t = \pi_i)$ that correspond to each state π_i. The estimated emission at time *t*, denoted l_t, only depends on the type of instruction executed at time *t*. As indicated previously, the inserted instruction type is the same as the regular instruction type, and thus attackers cannot estimate the emission probability for those states that execute the inserted instructions simply by observing the side-channel traces *l*.

We mounted simulated attacks to verify our analysis. The experiment results show that the random instructions inserted into the AES code executed on the ERIST-ARM7 core cannot be removed by the cross-correlation and HMM attacks proposed in [10].

C. Simulated CPA Attack Results

Correlation power analysis (CPA) [18] is one of the most popular SCAs. For this paper, we established a CPA attack platform using MATLAB and C language, and performed a simulated CPA attack against the standard AES-128 encryption algorithm executed by the ERIST-ARM7 core.

As a reference benchmark for our experiment conditions, we presented an attack against the same AES implementation on an original ARM7 core. Moreover, for better comparison, we created a softRIJJD-ARM7 model in RTL according to the method in [16] and performed the same CPA attack against it. The instruction insertion algorithm for softRIJJD was very similar to the plain uniform method.

The parameters for each method were chosen appropriately for the same performance overhead across the methods. The overheads were small so that the attacks could be successful with a reasonable number of traces. For ERIST, we used the parameters $a = 11$, $b = 3$, $k = 3$, and $d = 8$. For softRIJJD, we used the parameters $N = 3$ and $D = 5$.

In our attack experiment, there were approximately 1,745 clock cycles from the synchronization point to the attacked point when the AES code was performed on the original ARM7. After execution on softRIJJD-ARM7 or ERIST-ARM7, random delays were introduced. The statistical distributions of the delays introduced in each platform are shown in Figure 8. As can be seen, the delay variance introduced by ERIST is much greater than that introduced by softRIJJD with the same time overhead, which is approximately 21% (366 cycles).

The CPA attacks were mounted in the Hamming weight leakage model against the first AES key byte. The intermediate values generated in the first SubByte operation of the first encryption round were the attack target. Figure 9 shows the correlation coefficients for all the 256 key hypotheses in the attack point on each execution platform. It can be seen that to get significant peaks at the correct key hypothesis (0x03), the adversary needs 50, 3500, and 40000 power traces for statistical analysis, respectively.

Figure 10 shows the correlation coefficients for all key hypotheses, depending on the number of traces used in the attack. Recovering the 8-bit AES subkey for ARM7 (with the first-order success rate close to 1, similarly hereinafter) requires approximately 40 traces, and recovering softRIJJD-ARM7 requires approximately 3,000 traces. ERIST-ARM7 requires more than 35,000 traces to recover the 8-bit first key byte. Therefore, our method is significantly more secure, even for small delay durations and for a small number of delays. Table III lists detailed experiment results.

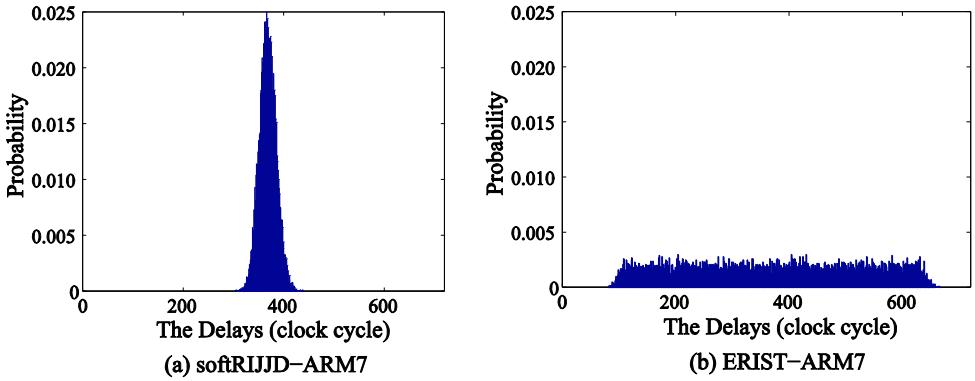

Fig. 8. Delay distribution introduced by two methods in attack experiment (with 10,000 measurements).

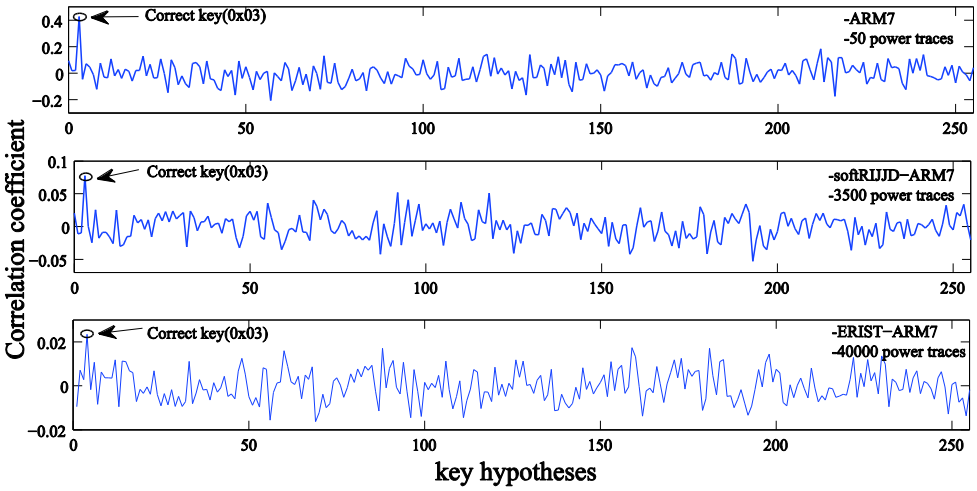

Fig.9. Correlation coefficients for all the 256 key hypotheses in the attack point on each execution platform

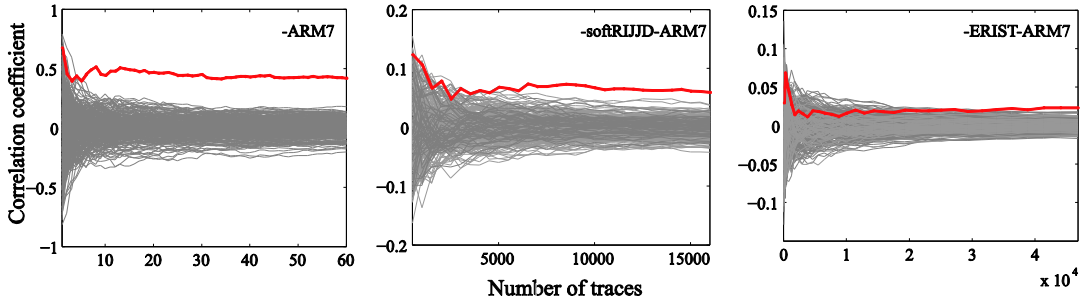

Fig. 10. Correlation coefficients with increasing number of power traces used in the attack for all key hypotheses (the trace for the correct hypothesis is highlighted).

TABLE III
DETAILED EXPERIMENT RESULTS FOR DIFFERENT METHODS

Platform	Required gates	Hardware overheads	Runtime Overheads	Required traces
ARM7	30081	0	0	40
softRIJJD-ARM7	31986	6.33%	20.86%	3000
ERIST-ARM7	32494	8%	21.09%	>35000

It must be noted that ERIST needs more hardware area than softRIJJD, mainly because an additional RIICR is used in ERIST, which obviously increases the usability and flexibility of our method.

V. CONCLUSIONS

We proposed a framework for a randomized instruction insertion technique in hardware that can automatically insert random real instructions into a cryptographic execution flow. The proposed framework can resist the latest signal processing and modeling attacks. Furthermore, we proposed an instruction insertion method for controlling the generation and insertion of the random instructions, which significantly improved the efficiency of our countermeasure. Theoretical analysis and experiment results showed that our countermeasure is significantly more secure and efficient than previous similar solutions.

REFERENCES

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology, CRYPTO99*, Springer, 1999, pp.388–397.

[2] S. Mangard, E. Oswald, T. Popp, *Power analysis attacks: revealing the secrets of smart cards*. Springer Science & Business Media, 2008.

[3] M. Barbosa and D. Page, "On the automatic construction of indistinguishable operations," in *Cryptography and Coding*, Springer, 2005, pp.233–247.

[4] C. Clavier, J.S. Coron, and N. Dabbous, "Differential power analysis in the presence of hardware countermeasures," in *Cryptographic*

Hardware and Embedded Systems, CHES 2000, Springer, 2000, pp.252–263.

[5] M. Rivain,and E.Prouff, "Provably secure higher-order masking of AES," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, Springer Berlin Heidelberg, 2010, pp. 413-427.

[6] N. V. eyrat-Charvillon, M. Medwed, S. Kerckhof, et al, "Shuffling against side-channel attacks: A comprehensive study with cautionary note," in *Advances in Cryptology–ASIACRYPT 2012*, Springer Berlin Heidelberg, 2012, pp.740-757.

[7] M. Tunstall and O. Benoit, "Efficient use of random delays in embedded software," in *Information Security Theory and Practices. Smart Cards, Mobile and Ubiquitous Computing Systems*, Springer, 2007, pp.27–3.

[8] S. R. Eddy, "Hidden Markov models," *Current opinion in structural biology*, vol.6, no.3, pp.361-365, 1996..

[9] M. F. Hashmi, A. R. Hambarde, A. G. Keskar, "Robust Image Authentication Based on HMM and SVM Classifiers." *Engineering Letters*, vol. 22, no. 4, pp.183-193, 2014.

[10] F. Durvaux, M. Renauld, F.X. Standaert, et al., *Efficient removal of random delays from embedded software implementation using hidden Markov models*. Berlin Heidelberg: Springer, 2013.

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

[12] Y. Lu, M. P. O'Neill, and J. V. McCanny, "FPGA implementation and analysis of random delay insertion countermeasure against dpa," in *ICECE Technology, 2008. FPT 2008*, IEEE, 2008, pp.201–208.

[13] Y. Lu, M. O'Neill, J.McCanny, "Evaluation of random delay insertion against DPA on FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol.4, no.1, 2010.

[14] D. May, H.L. Muller, and N.P. Smart, "Non-deterministic processors," in *Information Security and Privacy*, Springer, 2001, pp.115–129.

[15] P. Grabher, J. Großschädl, D. Page, "Non-deterministic processors: FPGA-based analysis of area, performance and security," in *Proceedings of the 4th Workshop on Embedded Systems Security*. ACM, 2009, pp.1-10.

[16] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "Randomized instruction injection to counter power analysis attacks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol.11, no.3, pp.69-78, 2012.

[17] S. Mangard, "Hardware countermeasures against dpa–a statistical analysis of their effectiveness," in *Topics in Cryptology–CT-RSA 2004*, Springer, 2004, pp.222–235.

[18] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems-CHES 2004*, Springer, 2004, pp.16–29.