RIJID: Random Code Injection to Mask Power Analysis based Side Channel Attacks

Jude Angelo Ambrose Roshan G. Ragel Sri Parameswaran
University of New South Wales, Sydney, Australia
{ajangelo,roshanr,sridevan}@cse.unsw.edu.au

ABSTRACT

Side channel attacks are becoming a major threat to the security of embedded systems. Countermeasures proposed to overcome Simple Power Analysis (SPA) and Differential Power Analysis (DPA), are data masking, table masking, current flattening, circuitry level solutions, dummy instruction insertions and balancing bit-flips. All these techniques are either susceptible to multi-order side channel attacks, not sufficiently generic to cover all encryption algorithms, or burden the system with high area cost, run-time or energy consumption.

A HW/SW based randomized instruction injection technique is proposed in this paper to overcome the pitfalls of previous countermeasures. Our technique injects random instructions at random places during the execution of an application which protects the system from both SPA and DPA. Further, we devise a systematic method to measure the security level of a power sequence and use it to measure the number of random instructions needed, to suitably confuse the adversary. Our processor model costs 1.9% in additional area for a simplescalar processor, and costs on average 29.8% in runtime and 27.1% in additional energy consumption for six industry standard cryptographic algorithms.

Categories and Subject Descriptors
I.5.2 [Design Methodology]: Feature evaluation and selection, Pattern analysis

General Terms
Security, Design, Measurement

Keywords
Side Channel Attack, Random Instruction Injection, Pattern Matching, Cross Correlation, Power Analysis

1. INTRODUCTION

As security concerns in embedded systems escalate, researchers increasingly try to find superior countermeasures to combat Side Channel Attacks (SCAs). Adversaries find properties such as power usage [12], processing time [4] and electro magnetic(EM) emissions [21], to correlate these external manifestations with internal computations. These properties are used to obtain critical information, such as a secret key of a secure application. Power analysis has been the most effective technique to extract secret keys during the execution of cryptographic algorithms using SCAs. [12, 15, 17].

Two types of power analysis attacks are used: (1) Simple Power Analysis (SPA); and (2) Differential Power Analysis (DPA) [20]. SPA involves the identification of computations and instructions used in a system by analyzing the power wave observed. DPA attacks are much more powerful than SPA attacks, as statistical analysis is used on the observed power wave to find secret keys [18, 20]. Typically, the adversary observes the power dissipated, and tries to identify specific power wave segments of corresponding rounds in the encryption of data within cryptographic programs [20]. Typical examples of such cryptographic programs are: AES and TripleDES.

For both SPA and DPA to be successful, it is imperative that the adversary is able to identify the power waveform with encryption rounds. If one can foil the identification of the power waveform, then the system becomes more secure against power wave based side channel attacks.

In this paper, for the first time, we introduce a hardware software Randomized Instruction inJectIon methoD, ***RIJID***, which scrambles the power wave so that the adversary is unable to identify specific segments such as encryption rounds within the entire power wave. ***RIJID*** prevents both SPA and DPA attacks by foiling the adversaries' attempts to identify power waves corresponding to encryption rounds. It is worth noting that our method uses $real$ instructions at random places (and NOT dummy instructions like NOP at fixed places) for the injection.

Since the processor has a pipeline, the power waveform at any point in time will be contributed to by a number of instructions. Even if the adversary somehow can remove the points in the power waveform corresponding to inserted instructions, it would be impossible from the resulting waveform to find out what the original power waveform would have looked like. Since the random bit flips within the random instructions would have corrupted the waveform to such an extent that statistical correlation for DPA is not possible. In fact it is even difficult to find the segments corresponding to the sbox rounds in the waveform.

To observe how close is the obfuscated sequence to a random sequence we have defined a new index called the ***RIJID*** index which uses cross-correlation to give us a measure of obfuscation. As far as we are aware, this is the first time a measure of this type has been proposed. This measure allows us to quickly find the level of obfuscation needed. To be absolutely certain, one must take electrical power measurements and try to perform DPA on it, which would take a very long time indeed.

The rest of the paper is organized as follows. Section 2 investigates previous research on side channel attacks, and analyzes previously proposed countermeasures. The key techniques and methodologies are presented in Section 3. The design flow of ***RIJID*** is explained in Section 4. Section 5 explains the important measurements used in this paper. Section 6 explains the experimental setup used for measurement and analysis. Results are shown in Section 7. Finally, the paper is concluded in Section 8.

2. RELATED WORK

Countermeasures proposed by researchers to prevent power analysis attacks could be divided into different categories such as masking, non-deterministic processing, current/power flattening, balancing and circuitry level solutions.

Kocher et. al. [20] proposed a masking technique to add noise into power lines during measurements, where the adversary needs to acquire more samples for a successful attack. Masking a computation or an intermediate result (data masking [15]) using random arbitrary values or functions combining the actual data, is a well known countermeasure. A random value is used with the actual secure computation to confuse the adversary such that wrong data values are predicted [5, 7, 24]. The Duplication Method [10] and Table masking [8] are similar techniques, dividing the standard *sbox* table into multiple different tables, where random values are used for computations. Masking can also be done to specific critical instructions by replacing them with secure special instructions [23]. Constant execution path or designing a piece of code to always yield the same result [3, 20] is another masking technique, where the adversary will not be able to predict the computations happening inside the system.

Non-Deterministic Processor [13], which uses random selection circuitry, is used to perform random issuing of independent code segments during runtime. This technique is also one of the better countermeasures, where the adversary cannot predict the instructions if they are executed out-of-order. Irwin et. al. [11] presents a software and hardware technique for non-deterministic processors, which uses an additional pipeline stage which performs random operations without modifying the effective data. A random register renaming technique [14] is proposed for the non-deterministic processor designed in [13], which uses

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. DAC 2007, June 4-8, 2007, San Diego, California, USA Copyright 2007 ACM 978-1-59593-627-1/07/0006...5.00

a logic circuit to rename the internal registers randomly, depending on the availability, to hide information leaks from secret key computation.

A current flattening technique is proposed by Muresan and Gebotys [16] to flatten the power wave of a processor.

The secure coprocessor[25], which is designed for AES-based biometric applications, uses a constant power dissipating logic for any bit transitions. This coprocessor area cost is 3X and power cost is 4X. A signal suppression technique is proposed by Ratanpal et. al. [22] where a special circuitry is designed to suppress the current dissipated by the processor.

A number of researchers have stated that the insertion of dummy instructions (NOPs) could be a solution to protect systems from side channel attacks [2, 20], though none to our knowledge have been implemented. Several dummy operation insertion techniques are proposed for ECC cryptosystems to create a constant execution path[3, 9]. Clavier et. al. [6] proposed an improved DPA attack called Sliding Window DPA (SW-DPA), to bypass the dummy instruction insertion technique.

As opposed to previous methods, ***RIJID*** provides a generalized solution with little human intervention compared to the masking methods [5, 7, 8, 10, 24], allowing the processor to take care of masking. On a Simplescalar processor, the additional area cost is just 1.98% compared to the area cost of constant logic chips [25], with an average energy cost of 27.1% and an average runtime cost of 29.8% compared to current flattening[16], for six industry standard benchmarks. ***RIJID*** confuses the adversary without flattening the current[16], but scrambling the patterns in the power wave. It can be applied to any vulnerable segment (and is not algorithm dependent like masking techniques [8, 24]). Dummy instruction insertions can be eliminated using simple time shifting[6], whereas ***RIJID*** injects real instructions at random places a random number of times. Hence the adversary will observe different power profiles on different tries.

Our Contributions are: (1), For the first time a hardware/software based randomized instruction injection technique is implemented to insert random $real$ instructions at random places to scramble the power dissipated by the processor; and (2), a simple mathematical formulation is introduced (called ***RIJID Index***) based on cross correlation that measures the scrambling provided by our technique. Note that this should not be used to compare two dissimilar techniques.

The Limitations of our approach are: (1), ***RIJID*** needs compiler support; (2), ***RIJID*** is proposed as a design time technique, since ***RIJID*** needs hardware changes; and, (3), we assume that our system is self contained with memory on chip.

3. RIJID FRAMEWORK

In this section, we present an overview of the hardware architecture for the ***RIJID*** framework and the software instrumentation, which is necessary for starting and stopping the randomization process.

3.1 Software Instrumentation

The software instrumentation for ***RIJID*** is performed at compile time. Figure 1 shows the encryption block of a cryptographic application which has several similar instruction segments (*sbox* rounds). Two flag instructions (*SET-FLAG* and *RESET-FLAG*) are inserted at the start and the end of the block as shown in Figure 1, to indicate to the processor the points to start and stop inserting random instructions at random places. When the processor fetches a *SET-FLAG* instruction, it starts generating random instructions and stops when it reads the *RESET-FLAG* instruction.

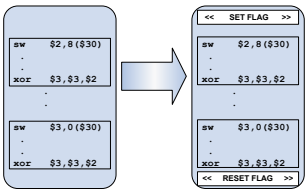


Figure 1: Software Instrumentation at Compile Time

Multiple occurrences of a loop containing a set of instructions, which causes a repeat power sequence template, is a potential place for flag instruction insertion. Recent SPA and DPA attacks reveal that the encryption block which has *sbox* rounds, is the most critical block [18], which is selected for random instruction injection in our technique. Programmers decide the blocks upon which to apply flag instructions. However, instrumentation can be automated for any given code, by having a parser which identifies encryption loops.

3.2 Hardware Architecture

Figure 2 shows a block diagram of the hardware architecture of ***RIJID*** framework. When the processor fetches the *SET-FLAG* instruction, a special register (*Flag*) is set. The dotted box in Figure 2 shows the hardware component added for ***RIJID*** framework. When *Flag* is set, the Random Generator component(*R/G*) sends a hold (*PC Hold'*) signal to the Program Counter (PC) and starts generating random instructions at random intervals. *PC Hold'* from the *R/G* is *ORed* with the *PC Hold* signal from the controller (*CONTR.*) before it is connected to the *PC*. The random instruction generation performed by *R/G* is limited by a pair of boundary values, which are set by the *SET-FLAG* instruction. This pair, which we call the *injection pair* represents: (1) the maximum number (*N*) of random instructions to be injected between two regular instructions when the flag is set; and (2) the maximum number (*D*) of regular instructions to be skipped before each injection.

PC Hold' signal from *R/G* is switched between on and off for random intervals based on the *injection pair*. When *PC Hold'* is high, the random instructions (*R/I*) generated by *R/G* are sent to the data port of the instruction register (*IR*). During hold, instruction which is pointed to by the PC is refetched from the instruction memory; however, this instruction is not written into the *IR*.

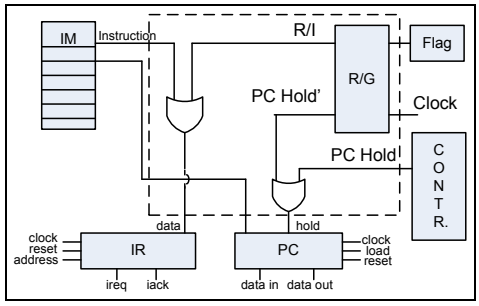


Figure 2: Random Instruction Injection

Since, execution of an instruction will generally affect the state of the processor, creating any random instruction will overwrite or edit effective data values. Therefore, only a limited set of instructions is selected such that, a random register is used and computed with the *zero* register and the result is written onto the same random register. Since the *R/G* selects random instructions from a specific set, it is called a pseudo random generator. For example, a randomly selected ADD instruction adds the value in the random register and the zero register and writes the result back to the same random register. Three different schemes of inserting random instructions were tried: (1),Instructions with zero registers; (2),instructions with *zero* and a fixed register; (3),instructions with *zero* and a randomly selected register. The insertions from (1) and (2) can be identifiable on the power wave, when same instructions are scheduled due to the lower amount of bit flips, which causes low power variation. The third scheme (use of random registers for consecutive random instructions) was the most appropriate for ***RIJID*** as it caused higher power variation due to bit flips in registers [14].

4. DESIGN FLOW

4.1 Software Design Flow

Figure 3(a) depicts the software design flow of the ***RIJID*** framework. The source code of a cryptographic application is compiled with the front-end of a compiler to generate the assembly version of the application (*.s files*). Critical blocks (blocks with encryption activities) in the assembly code is instrumented as explained in section 3.1. The resulting assembly file is assembled and linked to generate the binary of the application (iBinary in Figure 3(a)). Even though the instrumentation process here is performed manually, it is possible to write a software parser to perform this automatically as explained in section 3.1.

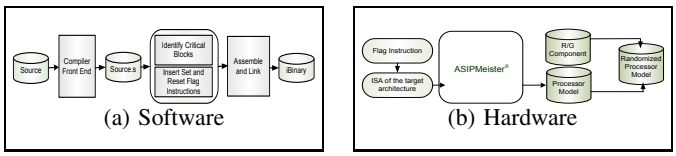


Figure 3: Design Flows

4.2 Hardware Design Flow

Figure 3(b) depicts the generation of a processor model which implements the ***RIJID*** framework. An additional flag instruction, which is used as a tag to enable and disable randomization, is combined into the instruction set architecture (ISA) of the target architecture. The new instruction is designed such that it sets and resets a flag when it is executed. This flag is used by the randomized instruction injector to manage the start and stop of random instruction injection as explained

in section 3.2. Combined ISA is then passed into an automatic processor design tool (*ASIPMeister*[1]) in Figure 3(b) to generate the ***RIJID*** processor model.

The output of ASIPMeister is a synthesizable VHDL processor model, which was enhanced by the *R/G* component (functional unit) as explained in section 3.2. *R/G* component is designed separately and then is combined with the processor.

5. RIJID INDEX

In our framework, $RIJID$ index is used as a measure to evaluate the randomization provided by random code injection to scramble a repeating pattern(*template*) in a power sequence. This is a simple mathematical method, where the vulnerability of a power sequence can be quickly predicted using $RIJID$ index instead of taking practical measurements and analyzing the electric waves. Our framework: (1), analyzes the original power sequence and extracts a template; and (2), uses $RIJID$ index as a measure to compute the randomization provided in the scrambled power sequence.

When a single occurrence (a template) of a repeating sequence is cross correlated [19] with the original sequence, significant peaks will appear in the output at places where the template matches with the original sequence.

When the significant peaks are removed from the cross correlated wave(called top elimination) the resulting mean is moved by a certain amount. Such mean movement for the cross-correlated sequences between the template and both the obfuscated sequence and the random sequence is less compared to the movement in original sequence, because there are no significant peaks. Since the random sequence does not have any correlation with the template, the random sequence has less movement than obfuscated sequence

The same number of significant peaks (decided using the template and the original sequence) are removed in all three sequences (Original, Random and Obfuscated) and the mean movement differences are used to find the ***RIJID*** index. ***RIJID*** index will give us a measure of how much the vulnerable sequence with template is obfuscated compared to a random sequence.

$$\Psi_{f,g} = \frac{\sum_{i=1}^{2N-1}(f \star g)_i}{(2N-1)} \tag{1}$$

$$\varphi_{f,g,T} = \frac{\sum_{i=1}^{2N-1}(f \star g)_i - \sum_{1}^{T} TopT}{(2N-1)-T} \tag{2}$$

$$\Delta_f = \Psi_{f,g} - \varphi_{f,g,T} \tag{3}$$

Equation (1) gives the mean of the cross correlation sequence of two sequences f and g. The number of points in the cross correlated sequence are $2N-1$, where N is the maximum number of points within the sequences which cross correlate. Equation (2) gives the mean value of the resulting cross correlation sequence with a number of peaks or maximum values removed. $TopT$ represents T number of maximum values in sequence $(f \star g)_i$. Equation (3) defines the mean movement (difference between the mean before and after top elimination) of a sequence f.

$$RIJID\ index = (\Delta_o - \Delta_z)/(\Delta_o - \Delta_r) \tag{4}$$

RIJID index ($0 \leq$ RIJID index ≤ 1), as defined in Equation (4), uses the mean movements of the Original (Δ_o), Obfuscated(Δ_z) and Random(Δ_r) sequences. The original sequence is used to form a related measure, where mean movement differences between the original sequence with obfuscated and random sequences are used. RIJID index reaches the value of one when the mean movement of the obfuscated sequence equals the mean movement of the random sequence. Such case gets the best scrambling from the $RIJID$ processor, where the dissipated power sequence appears as a random sequence(that is, no expected templates exist). The higher the RIJID index of a power sequence, the higher the masking.

6. EXPERIMENTAL SETUP

In this section, the main components used for experimentation and the process of randomization measurement is explained. The ***RIJID*** framework is implemented in a processor with PISA (Portable Instruction Set Architecture) instruction set (as implemented in SimpleScalar™ tool set with a six stage pipeline) processor without cache. Figure 4 shows the process of measuring ***RIJID*** index, using the original and scrambled code. Programs in C are compiled using GNU/GCC® cross compiler for the PISA instruction set. Original binary (oBinary) is produced from the original code of the program, which does not have the randomization technique applied. Instrumented binary (iBinary) is produced as explained in Section 4.1. ASIP Meister[1], an automatic ASIP design tool is used to generate synthesizable VHDL description of the processor as explained in Section 4.2.

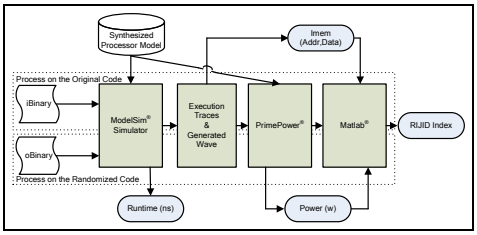


Figure 4: The Process of Measurement on Randomization

The same set of operations are performed for both the original processor with the non instrumented original binary (oBinary) and the ***RIJID*** enabled processor using the instrumented binary(iBinary). Corresponding binaries and synthesized processor models (Synopsys Design Compiler® is used for synthesis) are simulated together in ModelSim ® hardware simulator, which generates the stimulus wave with switching information. Using ModelSim® simulator, the execution trace is verified and extracted for future use. The runtime of each execution is also measured using ModelSim® simulator. The power values are measured using PrimePower® which gives the measurements in watts(W). The address($Addr$) and instruction opcode ($Data$) of instruction memory ($Imem$) are extracted from the execution trace as shown in Figure 4. Perl scripts are used to combine the Imem (Addr,Data) and power values(*Power*) taken from PrimePower® , which helps to map the power values for each instruction of the program execution. Matlab® is used to analyze and plot the combined data. ***RIJID*** index from the power sequences is calculated by implementing necessary functions in Matlab® .

The experiments demonstrated in this paper are performed for the cryptographic applications implemented in C language. A detailed explanation of the experiment on the TripleDES application is presented and the results of other applications are tabulated. Dissipated power waves are observed by running the encryption programs for specific keys and different data.

7. RESULTS

7.1 RIJID index Vs Runtime Overhead

Figure 5 shows the variation of RIJID index and runtime overhead for TripleDES program, when the *injection pair* (N,D) is varied. As expected, the runtime increases for each case when N increases, and decreases when D increases for each N.

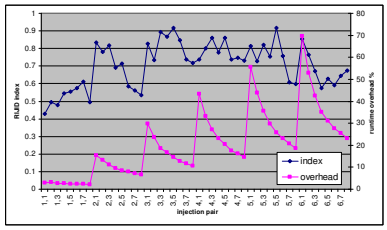


Figure 5: RIJID index and runtime overhead on TripleDES

The maximum RIJID index values from Figure 5 are produced for the injection pair (3,5) and (5,5) of values 0.9158 and 0.9157 with runtime overheads of 14.5% and 25.9% respectively. When the (N,D) pair is small, the possible combinations of injections are small and similar patterns may start to appear after several tries. Therefore we decided to have (N,D) = (5,5) as the most suitable injection pair as it will produce more permutations than (N,D)= (3,5).

Equation 5 shows the lower bound of the power pattern permutations of a single TripleDES round (which has the original size of 115 instructions, and six is the number of available instruction types which are randomly injected). The lower bound number is due to the fact that we have not taken into account the random register values and random registers used in the injected instructions.

$$C = (\sum_{k=1}^{N} \frac{6!}{(6-k)!})^{\frac{115}{D}} \tag{5}$$

The possible number of waveforms for a single round of TripleDES is approximately 2^{240} when using (N,D)= (5,5) and 2^{170} when using (N,D)= (3,5).

The other injection pairs ((N,D) $>$ (6,8)) are not tested since they will increase overhead without providing far greater scrambling.

7.2 RIJID on other cryptographic applications

RIJID is applied to different encryption programs and the results are tabulated in Table 1, where column 1 and 2 gives the N, D pair, Columns 3, 5, 7 gives the *RIJID* index for RSA, IDEA, and RC4 algorithms, and columns 4, 6, 8 and refer the runtime overheads. The results show that RSA gets the highest *RIJID* index when using (N,D)= (3,3) with a runtime overhead of 32.97% as injection pair, and (5,5) for IDEA and RC4 with a runtime overhead of 12.16% and 26.45%, respectively. Only the injection pairs of interest(from (3,3) to (6,6)) are shown in Table 1.

N	D	RSA		IDEA		RC4	
		index	% OH	index	% OH	index	% OH
3	3	0.9998	32.97	0.7204	8.5	0.9908	24.67
4	4	0.9896	15.88	0.7475	10.6	0.9523	26.25
5	5	0.9511	6.38	0.9738	12.16	0.9998	26.45
6	6	0.9607	11.76	0.7571	12.58	0.9364	24.80

Table 1: Runtime Overheads and *RIJID* index for cryptographic programs

The injection pair of (5,5) is considered to be the best choice to implement *RIJID* in all of these three applications, since it scrambles sufficiently, without too much overhead.

7.3 Energy Overhead

The average energy values for the original and scrambled (with Injection Pair 5,5) encryption blocks for different encryption programs are tabulated in Table 2. Since the maximum scrambling takes place using an Injection Pair value of (5,5) as presented in Table 1, the average energy values are calculated for (5,5) and the overheads are decided with the values of original encryption block.

	RSA (μJ)	RC4 (μJ)	IDEA (μJ)	T-DES (μJ)	Rijndael (μJ)	Blowfish (μJ)
Original	0.96	23.0	5.8	13.1	20.7	475.7
Obfuscated(5,5)	1.08	30.8	6.9	15.6	22.2	701.4
Overhead(%)	12.5	38.2	18.9	19.1	7.2	47.4

Table 2: Energy Overhead for Injection Pair (5,5)

In all cases energy increases slightly, attributable to the increase in runtime.

7.4 Hardware Overhead

An additional functional unit is created for the random instruction generation (R/G as explained in Section 3.2). Our *RIJID* processor costs an additional 1.98% area overhead when compared with the normal processor without *RIJID*, as shown in Table 3. The clock period remains the same for the *RIJID* processor as the original processor.

	Area ($cell$)	Clock Period (ns)
Normal Processor	111,188	41.33
RIJID Processor	113,393	41.33
Overhead(%)	1.98	0

Table 3: Hardware Overheads

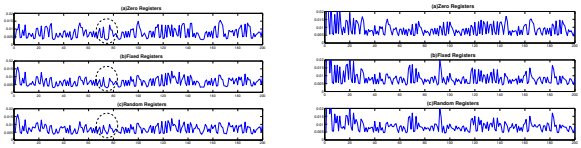
RIJID provides less hardware overheads when compared to other hardware designs, such as the secure coprocessor proposed by Tiri et. al. [25] which costs three times increase in area.

7.5 Varying the Instruction Injection

Figure 6(a) depicts the variation of power values when using the same random instruction injection, but using: 1) immediate operands in the random instructions which are set to zero; 2) registers which do not change during the injection for a given random instruction (thus, injected ADD will always work with a fixed register); and 3) random registers with random immediate operand in the injected instructions.

When similar instructions are injected next to each other (circled in the Figure 6(a)) due to the values scheduled by the random generator, a constant power tends to be seen for the zero register and fixed register cases. This is because of insufficient bit flips inside registers to cause a significant change in the power consumption. The constant power is changed by using random register writes as shown in the figure and by causing random bit flips inside registers. This figure is a simple demonstration to show that the greater the randomization the better.

Figure 6(b) shows the power profile for the same instruction segment as the segment shown in Figure 6(a) with a different seed value. As from the figures, the power profile significantly varies when changing



(a) Seed 1 (b) Seed 2

Figure 6: Register Types on Injected Instructions

the seed of the random generator for the random register case. The seed is designed to be changed every time the program executes. Therefore the adversary will observe different power profiles for different tries which makes it harder to determine the injected instructions.

8. CONCLUSIONS

This paper proposes a randomized instruction injection technique (*RIJID*) to prevent side channel attacks. Random number of real instructions at random places are injected during the runtime of the processor to scramble the power wave, so that adversaries cannot extract any useful information by observing the power wave leakage from the processor.

A new pattern matching methodology is used to measure the degree of randomization to identify suitable injection values. The *RIJID* processor consumes an area overhead of 1.98%, an average runtime overhead of 29.8% and an average energy overhead of 27.1%.

Our technique can be used to prevent several side channel attacks such as Power Analysis (SPA and DPA), Electro magnetic analysis (SEMA and DEMA). Future work includes designing a signature-based smart processor for the random instruction injection to avoid software instrumentation and a hardware logic to execute the independent code segments using the injected random instructions.

9. ACKNOWLEDGEMENT

The Authors would like to thank Alex Ignjatovic, Jorgen Peddersen and Jeremy Chan for their great help in experiments. And we also like to thank Australian Research Council for their funding to support this research (project number DP0559880).

10. REFERENCES

[1] The PEAS Team. ASIP Meister, 2002. Available at: http://www.eda-meister.org/asipmeister.
[2] M.-L. Akkar, R. Bevan, P. Dischamp, and D. Moyart. Power analysis, what is now possible... In *Asiacrypt '00*, pages 489–502, London, UK, 2000. Springer-Verlag.
[3] M. Barbosa and D. Page. On the automatic construction of indistinguishable operations. In *Cryptography And Coding*, pages 233–247, November 2005.
[4] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX*, August 2003.
[5] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, pages 398–412, 1999.
[6] C. Clavier, J.-S. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In *Ches '00*, pages 252–263, London, UK, 2000.
[7] J.-S. Coron and L. Goubin. On boolean and arithmetic masking against differential power analysis. In *Ches '00*, pages 231–237, London, UK, 2000. Springer-Verlag.
[8] C. Gebotys. A Table Masking Countermeasure for Low-Energy Secure Embedded Systems. *IEEE Trans. on VLSI*, 14(7):740–753, 2006.
[9] C. H. Gebotys and R. J. Gebotys. Secure elliptic curve implementations: An analysis of resistance to power-attacks in a dsp processor. In *CHES '02:*, pages 114–128, London, UK, 2003. Springer-Verlag.
[10] L. Goubin and J. Patarin. Des and differential power analysis (the "duplication" method). In *Ches '99*, pages 158–172, London, UK, 1999. Springer-Verlag.
[11] J. Irwin, D. Page, and N. P. Smart. Instruction stream mutation for non-deterministic processors. In *Asap '02*, page 286, Washington, DC, USA, 2002.
[12] S. Mangard. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In *icisc 2002*, volume 2587, pages 343–358, 2003.
[13] D. May, H. L. Muller, and N. P. Smart. Non-deterministic processors. In *Acisp '01*, pages 115–129, London, UK, 2001. Springer-Verlag.
[14] D. May, H. L. Muller, and N. P. Smart. Random register renaming to foil dpa. In *Ches '01*, pages 28–38, London, UK, 2001. Springer-Verlag.
[15] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Computers*, 51(5):541–552, 2002.
[16] R. Muresan and C. H. Gebotys. Current flattening in software and hardware for security applications. In *CODES+ISSS*, pages 218–223, 2004.
[17] S. B. Ors, F. Gurkaynak, E. Oswald, and B. Preneel. Power-analysis attack on an asic aes implementation. *itcc*, 02:546, 2004.
[18] E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In *ct-rsa 2006*, pages 192–207. Springer, 2006.
[19] Paul Bourke. Cross Correlation: AutoCorrelation – 2D Pattern Identification. url:http://astronomy.swin.edu.au/ pbourke/other/correlate/index.html, 1996.
[20] J. J. Paul Kocher and B. Jun. Differential Power Analysis. 1998. First article on DPA.
[21] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *E-smart*, pages 200–210, 2001.
[22] G. B. Ratanpal, R. D. Williams, and T. N. Blalock. An on-chip signal suppression countermeasure to power analysis attacks. *IEEE Transactions on Dependable and Secure Computing*, 01(3):179–189, 2004.
[23] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the energy behavior of des encryption. *date*, 01:10084, 2003.
[24] F.-X. Standaert, E. Peeters, and J.-J. Quisquater. On the Masking Countermeasure and Higher-Order Power Analysis Attacks. In *ITCC 2005*, pages 562–567, 2005.
[25] K. Tiri, D. Hwang, A. Hodjat, B. Lai, S. Yang, P. Schaumont, and I. Verbauwhede. A side-channel leakage free coprocessor ic in 0.18um cmos for embedded aes-based cryptographic and biometric processing. In *Dac '05*, pages 222–227, New York, NY, USA, 2005. ACM Press.