A Smart Random Code Injection to Mask Power Analysis Based Side Channel Attacks

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

ABSTRACT

One of the security issues in embedded system is the ability of an adversary to perform side channel attacks. Power analysis attacks are often very successful, where the power sequence dissipated by the system is observed and analyzed to predict secret keys.

In this paper we show a processor architecture, which automatically detects the execution of the most common encryption algorithms, starts to scramble the power waveform by adding randomly placed instructions with random register accesses, and stops injecting instructions when it is safe to do so.

Our technique prevents both Simple Power Analysis (SPA) and Differential Power Analysis (DPA). This approach has less overheads compared to previous solutions and avoids software instrumentation, allowing programmers with no special knowledge to use the system. Our processor model costs an additional area of 1.2%, and an average of 25% in runtime and 28.5% in energy overheads for 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, Signature Identification, Cross Correlation, Power Analysis

1. INTRODUCTION

Side Channel Attacks are a major concern for security experts of Embedded Systems due to the increasing usage of such systems in real life applications. An adversary observes properties like power [19], electro magnetic (EM) emission [29] and processing time[6] from the embedded processor when executing cryptographic programs. The observed property is then correlated with the structure of the program to predict secret keys. Simple power analysis (SPA) and differential power analysis (DPA) are the most popular attacks, and several researchers have shown [3, 26], that it is possible to extract secret keys from the observed power sequence using fewer power samples, than if one were to use a brute force method. Simple power analysis uses the direct relationship between the instructions executed and the power sequence by finding the places where instructions are executed in the dissipated power sequence. Secret Keys are computed based on the magnitude of power values at the identified places, for different data values [28]. The magnitudes can be used to measure the hamming weight (denotes the number of bits set to 1 during an instruction execution [5]), which is used to identify the secret key [23]. DPA, which is more powerful than SPA, uses statistical analysis, based on the principle that there is a significant power variation between manipulating 0's and 1's [28].

Cryptographic algorithms (such as AES, SHA, DES, TripleDES, Seal, RC4, RSA and ECC) encode plaintext into ciphertext using secret keys. Most of these algorithms encrypt data in multiple iterations (or rounds) using a secret key, where the ciphertext after an iteration is used as the plaintext for the next iteration. For example, AES [19] and DES [12] use SBOX (a fixed or a dynamic mapping table which returns values for ciphertext based on the plain text and decides the strength of the encryption algorithm) rounds, where the subkeys from the secret key are used in each round to produce the ciphertext. Since same instructions are executed in each iteration of the loop, the dissipated power produces similar patterns corresponding to each iteration [3]. Adversaries analyze these patterns (these patterns are referred as $templates$ from this point onwards) in the power sequence and attempt to predict which pattern corresponds to what iteration. Past DPA attacks articulate that the adversary needs specific critical segments in the power sequence to perform a successful DPA attack [12, 23, 25]. Such vulnerable power segments can easily be identified in the power sequence by analyzing the repetitive templates.

For both SPA and DPA to be successful, it is imperative that the adversary is able to identify the power waveform with critical segments (like 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.

To obfuscate the critical segments inside the encryption block of a cryptographic program, the processor can be modified to apply masking when it detects such critical segments. This can be implemented in two ways: (1), Software instrumentation, where programmer specifies tags (special instructions) in the code, where critical segments are; or (2), Defining signatures of these critical segments to identify such signatures at runtime. Signatures, defined based on the instructions executed, takes away from the programmer the responsibility associated with adding instructions in method (1). Since nowadays, the software of an application is developed by multiple programmers working in different modules and combining them together, we strongly believe that hardware modification does guarantee the secure feature been properly implemented.

Several researchers have implemented methodologies to find the frequently executed loops using signatures based on jump instructions [14, 22] for HW/SW co-design methods. Thus far, there has been no implementation of signatures to identify critical segments (such as encryption block) in a cryptographic program. Several secure operations (like random instruction injection proposed in this paper) could be automated by pre-defining such signatures inside the processor.

This paper presents a hardware Automatic Randomized Instruction inJectIon methoD (*A-RIJID*), which scrambles the power wave so that an adversary is not able to identify specific segments such as encryption rounds from the entire power wave. For the first time, a sig-

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

nature based system is implemented to identify the critical segments for the instruction injection. A simple mathematical formulation, *RIJID* index, is used to evaluate the scrambling provided by the random instruction injection. *A-RIJID* prevents both SPA and DPA attacks by foiling the adversaries' attempt to identify the power waves corresponding to the encryption rounds.

It is worth noting that we are not injecting dummy instructions(like NOPs which execute only fetch), but inject *real* instructions performing operations on all the pipeline stages (such as OR, AND etc.).

Since the processor has a pipeline, the power waveform at any point in time will be contributed to by a number of instructions. If sufficient random instructions are injected, 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.

The work in this paper, in contrast to previous obfuscation methods, modifies the processor itself to insert random instructions (not just NOPs, but a selection of instructions, with randomized operands to change hamming weights) by identifying the signatures for the critical blocks from the instruction execution on its own. These random instructions are fed through all pipeline stages foiling acoustic attacks and EM attacks.

Our technique, *A-RIJID*, completely eliminates the onus on the programmer to take care of obfuscation. Additionally, the area and energy consumption itself are smaller than previous countermeasures.

The rest of the paper is organized as follows. Section 2 investigates previous research on side channel attacks, and analyzes previous countermeasures proposed. The signature definitions are explained in Section 3. Section 4 explains the *A-RIJID* framework. The design flow of *A-RIJID* is explained in Section 5. Section 6 explains the key measure used in this paper. Section 7 explains the experimental setup used for measurement and analysis. Results are shown in Section 8. Finally, the paper is concluded in Section 9.

2. RELATED WORK

Over the years, there have been several hardware and software solutions proposed to counteract side channel attacks. The key solutions are masking, constant path execution, non-deterministic processing, current flattening, balanced logic, and dummy instruction insertion.

Masking a computation or an intermediate result (data masking [23]) using random arbitrary values or functions combined with 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 [7]. The Duplication Method [15] and Table masking [12] are similar techniques which divide 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 [31].

Constant execution path or designing a piece of code to always yield the same result [4, 28] is another masking technique, where the adversary will not be able to predict the computations happening inside the system. Several researchers proposed the insertion of dummy operations for ECC systems to make every execution path uniform [4, 13]. For example, if there is no add performed in an execution path, a dummy add is inserted in the code to create identical execution paths [4]. This will prevent the adversary from differentiating add and double operations in ECC.

Non-Deterministic Processor [20], which uses a random selection circuitry, is used to perform random issuing of instructions within the 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 [17] presents a software and hardware technique for non-deterministic processors, which uses an additional pipeline stage to perform random operations without modifying the effective data. A random register renaming technique [21] is proposed for the non-deterministic processor designed in [20], which uses a hardware logic 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 [24] to flatten the power wave of a processor. This technique modifies the source code by inserting NOPs to impose a certain amount of discharging to flatten the dissipated current.

The secure coprocessor[33], 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.[30] where a special circuitry is designed to suppress the current dissipated by the processor.

A number of researchers have stated that the injection of dummy instructions (NOPs) could be a solution to protect systems from side channel attacks [3, 28]. Clavier et al.[8] proposed an improved DPA attack, called the Sliding Window DPA (SW-DPA), to bypass the dummy instruction injection technique. Daemen and Rijmen [10] state that inserting dummy instructions is not an appropriate solution where each instruction has its own power profile and synchronization is possible (for pipelined systems, each instruction affects the power wave of its neighboring instructions). A random instruction masking technique, similar to our approach, is presented in [11] where they inject random pseudo shift instructions by tagging special instructions in the code to identify places for injection.

In general, data masking techniques have been vulnerable to second order DPA attacks [26, 34]. The table masking methods [12, 15] are algorithm specific approaches (works for the algorithms which use an *sbox*) and they successfully prevent DPA. However, masking techniques require a high degree of manual intervention and they fail to scramble the power patterns since the instructions executed remain unchanged. The dummy operation insertion techniques proposed for ECC systems [4, 13] are application specific and needs significant human intervention. The random instruction masking technique[11] needs modification of the source code to indicate where to insert random instructions and a modification of shift operations is also necessary(for example; instead of shifting eight places, use a loop to shift one place at a time and inject random instructions and then shift the other until you shift eight times). This technique focuses only on shift operations, preventing adversaries from detecting shifts. It also increases the code size by a significant amount, as all of the shift operations need to be masked (they block only a single sbox). However, there is a possibility that the adversary could still detect the other parts of the encryption block which are not masked. Non-deterministic processors [20] are not feasible in highly dependent software code, which cannot be executed *out-of-order*. The techniques proposed using non-deterministic processors [17, 21] have complex circuitry but do not have their overheads reported. The circuitry level solutions [30, 33] cost significant area and energy overheads. The signal suppression technique [30] does not completely prevent DPA, but tries to make the attack more difficult. The current flattening technique, which is considered the most appropriate countermeasure for power analysis based SCAs, increases execution time by up to 75%, and flattens locally, based upon basic blocks.

As opposed to the pitfalls from previous methods, *A-RIJID* provides a generalized solution with no human intervention compared to masking, constant execution path and current flattening methods [4, 7, 9, 12, 13, 15, 24], allowing the processor to take care of masking. On a processor with PISA (Portable Instruction Set Architecture) instruction set (as implemented in SimepleScalar™), the additional area cost is just 1.2% compared to the area cost of constant logic chips[33], with an average energy cost of 28.5% and an average runtime cost of 25% compared to current flattening[24], for industry standard benchmarks. *A-RIJID* confuses the adversary without flattening the current[24], but scrambling the patterns in the power wave. It can be applied to any vulnerable segment (and is not algorithm dependent like masking techniques [12, 32]). Dummy instruction insertions can be eliminated using simple time shifting[8] and the random instruction masking [11] injects random instructions at fixed places denoted

in the source code, whereas *A-RIJID* injects real instructions at random places a random number of times. Hence the adversary will observe different obfuscated power profiles on different tries.

Contributions

1. an implementation to automatically trigger random instruction insertion (at random places with random registers) when precomputed signatures are detected in encryption blocks.
   - we show that this automatic triggering has little or no impact upon non-encryption programs.
2. a simple mathematical formulation is used (called *RIJID Index*) based on cross correlation that measures the scrambling provided by our technique, which coarsely indicates the level of scrambling achieved. This is a relative measure which can be used to compare one power waveform against the other, which use similar techniques to scramble. Note that this should not be used to compare two dissimilar techniques.

Limitations

1. *A-RIJID* is proposed as a design time technique, since it needs hardware changes;
2. we assume that our system is self contained with memory on chip; and,
3. more signatures need to be added when an entirely different encryption algorithm with different nature is introduced, compared to the ones considered in this paper.

3. SIGNATURE IMPLEMENTATION

Signatures can be used to identify certain properties in a program at runtime. Such signatures should be defined using the instructions executed, allowing the processor to identify the pattern of instructions executed and respond accordingly. Cryptographic programs contain an encryption block which does secure computations using secret keys. The key aim of this paper is to identify such encryption blocks by defining an appropriate signature.

A *self-concomitance* analysis [18] is performed on the programs in Table 1 to identify the possible instructions for the signature of an encryption block, by analyzing a single instruction, and how quickly it is executed again within cryptographic parts of the program, then comparing this *self-concomitance* metric to the non-cryptographic parts of the program. Similarly, another technique called the *concomitance* analysis [18] is used to find the combination of instructions which could be used as a signature. Due to space restrictions, this analysis is not shown here, but the interested reader is referred to [18].

As per this analysis, the XOR instructions (repeated within a window of 85 instructions) are mostly used in encryption blocks of a cryptographic program. A multiplication then the division within a window size of five is another signature which stands out in RSA. Table 1 shows the number of $xor\ hits$ and Multiplication and Division hits for different benchmarks. The first column of Table 1 divides the benchmarks into General and cryptographic programs. The second column details the name of applications, the third column gives the number of XORs in the application, and the fourth gives the number of MULT and DIV instructions within five instructions of each other ($m\ hits$). The fifth column gives the total number of instructions in the trace of the program. The final two columns show the percentage of xor and $m\ hits$ which occur in the trace. The total xor and the $m\ hits$ were obtained using the SimpleScalar instruction set simulator.

The percentage $xor\ hits$ (% X) for general programs (non- cryptographic programs) are much less than for the cryptographic programs as shown in Table 1. All analyzed cryptographic programs, except RSA, have significant $xor\ hits$ due to the usage of XOR instructions for encryption. As Table 1 depicts, SHA has the maximum $xor\ hit$ percentage of 10%, while SEAL has 4%.

The $m\ hits$ within a window of five instructions is analyzed for the benchmarks as shown in Table 1. No other program except RSA has $m\ hits$. RSA has an $m\ hit$ percentage of 0.12% on the total number of instructions executed.

	Programs	xor hits	m hits	Inst.	% X	% M
G	Dijsktra	686	0	975333	0.00	0.00
E	JPEG	3	0	9167386	0.00	0.00
N	FFT	449	0	732776	0.06	0.00
E	QSORT	23	0	22684	0.10	0.00
R	BasicMath	8196	0	4489680	0.18	0.00
A	StringSearch	1125	0	300710	0.37	0.00
L	CRC32	51	0	11296	0.48	0.00
	Blowfish	26184	0	301753	8.67	0.00
C	SHA	1325459	0	13209078	10.03	0.00
R	Rijndael	265	0	13268	1.99	0.00
Y	SEAL	44279	0	1100640	4.02	0.00
P	RC4	312126	0	24882358	1.25	0.00
T	TripleDES	1060	0	30019	3.50	0.00
O	ECC	10288585	0	897392501	1.15	0.00
	RSA	1	8	7095	0.00	0.12

Table 1: Instruction hits on BenchMarks

Based on this analysis we define two different signatures: (1), to capture the encryption blocks which use XOR (such as Blowfish, SHA, Rijndael, SEAL, RC4 and TripleDES); and, (2), to capture the encryption blocks which use Multiplication followed by Division within a window size of five (for RSA).

3.1 Signature 1 : sigXOR

The first signature (*sigXOR*) is defined as shown in the diagram on the left side of Figure 1. When an XOR is executed for the first time, it is identified as the start of the signature. The signature expires when there is no more XORs seen before 85 instructions. The value of 85 is decided based on our concomitance analysis and the research of Ross and Vahid [14].

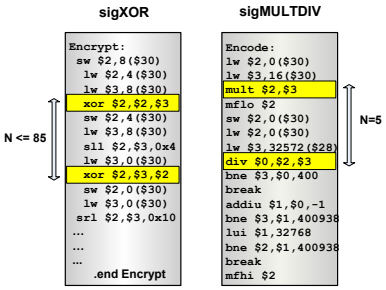

Figure 1: Signature Definitions

Therefore, an identification of an XOR instruction indicates the existence of a *sigXOR* signature. Multiple *sigXOR* detections are possible, where each new XOR occurrence after an expiry is considered the start of another *sigXOR*.

3.2 Signature 2 : sigMULTDIV

RSA algorithm is an exception which does not use XOR within its encoding block. Instead, it uses MULT and DIV instructions. Therefore a signature is defined as shown in the right side diagram of Figure 1, where the signature is detected when a MULT and DIV instructions are seen within five instructions. This signature is considered expired when no such signature is seen again before 85 instructions after the previous *sigMULTDIV* execution. According to our analysis, no program (in the tested set) other than RSA contains *sigMULTDIV* signature as shown in Table 2

Unlike *sigXOR* where the signature is started when XOR is executed, *sigMULTDIV* is started only when both MULT and DIV are seen.

4. A-RIJID FRAMEWORK

The *A-RIJID* Framework is shown in Figure 2 which includes the Random Generation (R/G) component to perform signature analysis and instruction injection. When CPU executes the XOR instruction, a special flag register (*XORSEL*) is set. Based on the value in $XORSEL$, R/G uses a $counter$ to identify the signature and sets the SEL flag. It also controls $XORSEL$ for reading and writing.

Two similar flags ($DIVSEL$ and $MULTSEL$) are used for MULT (Multiplication) and DIV (Division) instructions to identify the *sigMULTDIV* signature which is explained in Section 3. The R/G

sets and resets SEL flag based on the values inside $DIVSEL$ and $MULTSEL$ which are set by CPU, when the corresponding instructions execute.

When SEL is set, R/G sends a hold (*pc hold'*) signal to the Program Counter (PC) and starts generating random instructions at random intervals. The *pc hold'* from R/G and the *pc hold* signal from $controller$ are multiplexed together before they are connected to the PC. The generated random instruction (***R/I***) is multiplexed with the data bus ($Inst.$) from the Instruction Memory(IM). The multiplexers are selected by R/G based on the instruction injection.

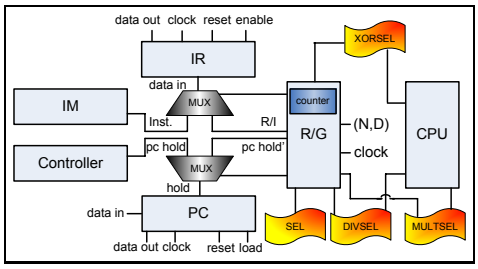

Figure 2: A-RIJID Architecture

The random instruction generation performed by R/G is limited by a pair of boundary values (Injection Pair:N, D), which are set inside R/G. N represents the maximum number of random instructions to be injected between two regular instructions. D represents the maximum number of regular instructions to be skipped before each injection.

The *pc hold'* signal from R/G is switched between on and off for random times 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***.

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. The use of random registers for consecutive random instructions was the most appropriate for ***A-RIJID*** as it caused higher power variation due to bit flips in registers [21].

When one signature is detected(*sigXOR*/*sigMULTDIV*), the system does not detect the other. Hence this implementation avoids nested combinations.

5. HARDWARE DESIGN FLOW

This section presents the hardware design flow, describing how the ***A-RIJID*** framework is implemented in a pipelined RISC processor. Figure 3 depicts the generation of a processor model which implements the ***A-RIJID*** framework. The ISA is fed into an automatic processor design tool (*ASIP Design Tool* in Figure 3 - ASIPMeister[2]) to generate the ***A-RIJID*** processor model.

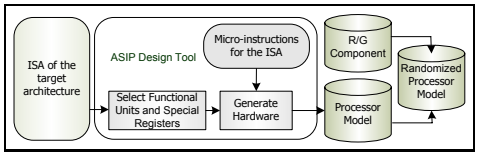

Figure 3: Hardware Design Flow

Necessary functional units and special registers for signature detection are selected. The micro-instructions and the functional units are combined to generate a hardware 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 4. ***R/G*** component is designed separately and then is combined with the processor.

6. RIJID INDEX

To observe how close is the obfuscated sequence to a random sequence, we have defined an index called the ***RIJID*** index which uses cross-correlation to give us a measure of obfuscation. This measure allows us to quickly find the level of obfuscation needed, instead of taking practical measurements and analyzing the electric waves. 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.

Our framework: (1), analyzes the original power sequence and extracts a repeating template and (2), uses ***RIJID*** index as a measure to compute the randomization provided in the scrambled power sequence.

When a single occurrence (the template) of a repeating sequence is cross correlated with the original sequence, significant peaks will appear in the output at places where the template matches with the original sequence [27]. The cross-correlation between the template and the random sequence does not produce a significant peak.

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 number of significant peaks are decided based on the number of occurrences of the template.

Similar cross-correlation curves are taken for both random sequence and the obfuscated sequence with the same template. The same number of significant peaks(decided using the template and the original sequence) are removed in all three sequences(Original, Random, Obfuscated) and the mean movement differences will be 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

		Normal(μs)	sigXOR (μs)	% O.H	sigMULTDIV (μs)	% O.H	A-RIJID (μs)	% O.H	% Energy O.H
G	**Dijsktra**	549697	549706	0.00	549697	0.00	549706	0.00	5.7
E	**JPEG**	4234937	4234997	0.00	4234937	0.00	4234997	0.00	5.7
N	**FFT**	5725	5824	1.70	5725	0.00	5824	1.70	7.6
E	**QSORT**	3285	3307	0.67	3285	0.00	3307	0.67	6.5
R	**BasicMath**	179097	181879	1.50	179097	0.00	181879	1.50	7.4
A	**StringSearch**	48412	49333	1.90	48412	0.00	49333	1.90	7.8
L	**CRC32**	35	40	15.80	35	0.00	40	15.80	20.8
C	**Blowfish**	36397	59692	64.00	36397	0.00	59692	64.00	72.8
R	**SHA**	3076	3226	4.80	3076	0.00	3226	4.80	10.9
Y	**Rijndael**	1752	1896	8.20	1752	0.00	1896	8.20	14.5
P	**SEAL**	152392	206409	35.50	152392	0.00	206409	35.50	42.2
T	**RC4**	4504	4572	1.50	4504	0.00	4572	1.50	7.3
O	**TripleDES**	2902	3941	35.80	2902	0.00	3941	35.80	42.6
	RSA	160	165	3.10	175	9.40	180	13.10	9.0

Table 2: Runtime Overheads for A-RIJID processors

sequence. Such case gets the best scrambling from the *A-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.

7. EXPERIMENTAL SETUP

In this section, the main components used for experimentation and the process of randomization measurement is explained. *A-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 obfuscated processors. Programs in C are compiled using GNU/GCC® cross compiler for the PISA instruction set and the binary is produced. ASIPMeister[2], an automatic ASIP design tool is used to generate a synthesizable VHDL description of the processor as explained in Section 5.

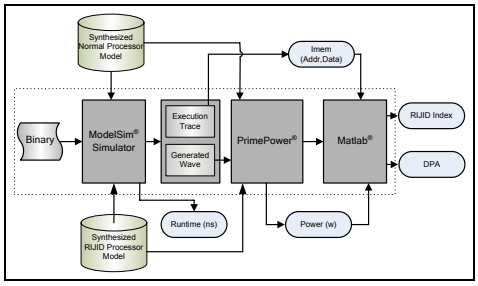

Figure 4: Measurement on Randomization

The binary 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. DPA is performed by analyzing the source code and the extracted power values using Matlab® .

Prior experimentation showed that an injection pair (see Section 4) of (N,D) = (5,5) provided sufficient obfuscation with a *RIJID* index which was above 0.7.

The experiments demonstrated in this paper are performed for the applications implemented in C, which are taken from MiBench suite [16] and Sourcebank[1] suite. Evaluation is performed on these programs using four different processors: (1), ***softRIJID* processor**, where tags are manually inserted in the assembly file to indicate the encryption block (this requires manual intervention, but allows a base processor to compare with); (2), ***sigXOR* processor**, identifying *sigXOR* signature; (3), ***sigMULTDIV* processor**, identifying *sigMULTDIV* signature; and (4), a **combined processor (*A-RIJID*)**, identifying both *sigXOR* and *sigMULTDIV* signatures.

8. RESULTS

8.1 Runtime & Energy

Table 2 depicts the runtime and energy overheads of *A-RIJID* processors for different benchmarks. The first column of Table 2 divides the benchmarks into General and Cryptographic programs. The second column details the name of applications, the third column gives the runtime of programs on $Normal$ processor (a general processor without any signature recognition). The fourth, sixth and eighth columns show the runtime of programs when *sigXOR*, *sigMULTDIV* and *A-RIJID* applied, and the fifth, seventh and ninth columns depict respective runtime overheads of specified processors. The final column shows the energy overhead of each benchmark when *A-RIJID* is applied.

The runtime overheads when using sigXOR is much larger for cryptographic programs compared to non-cryptographic programs as shown in Table 2. Blowfish has the highest runtime overhead of 64%, and the lowest is RC4, costing 1.5%, amongst the cryptographic programs. Even though CRC32 (costs 15.8% in runtime) is not categorized as a cryptographic program, it can be considered a vulnerable program, as it computes checksums using XOR instructions. Note that despite just having 0.48% of XOR instruction in the trace of the CRC application, the overhead is high, due to the fact that the XORs are close to each other, and are frequently executed. RSA has 3.1% in runtime overhead, which gets just one XOR hit as shown in Table 1. This XOR is outside the encryption block, yet due to RSA's small size, we get a large overhead. Table 2 shows that *sigXOR* does not significantly affect non-cryptographic programs (except CRC32) if we apply *A-RIJID*. The maximum runtime overhead for non-cryptographic programs is for StringSearch with just 1.9%. The *A-RIJID* consumes 13.10% of runtime when applied to RSA as shown in Table 2. All other programs except RSA does not have any *sigMULTDIV* hits, hence have no runtime overhead.

The energy overheads for benchmarks proportionally increases with runtime overhead as shown in Table 2, due to the small variation in dissipated power. Blowfish gives the maximum energy overhead of 72.8%, while TripleDES and SEAL dissipating 42.6% and 42.2%.

8.2 Hardware Summary

Table 3 depicts the hardware overheads of *A-RIJID* processors. The first column of Table 3 denotes the types of processors used. The second column states the area of each processor. The clock period for each processor is listed in the third column. The area overheads are presented in the fourth column. The processor area is smaller for *sigXOR*, *sigMULTDIV* and *A-RIJID* compared to *softRIJID* as shown in Table 3 because of no special instruction usage.

The clock period does not have any significant difference when signature recognition is implemented. *A-RIJID* costs an additional area of 1.2% which is higher than *sigXOR* (with 0.8%) and *sigMULTDIV* (with 0.9%), due to the combinational circuit of both *sigXOR* and *sigMULTDIV*. *A-RIJID* provides reduced hardware overheads when compared to other hardware designs, such as the secure coprocessor proposed by Tiri et. al. [33] which costs three times increase in area.

Processor	Area ($cell$)	Clock (ns)	Area Overhead (%)
Normal	111,188	41.33	N/A
softRIJID	113,302	41.47	1.9
sigXOR	112,123	41.69	0.8
sigMULTDIV	112,200	41.69	0.9
A-RIJID	112,545	41.69	1.2

Table 3: Hardware Summary

8.3 Obfuscation and the RIJID Index

The higher the *RIJID* index (explained in Section 6), the higher the scrambling provided and lower the vulnerability of the power sequence. Table 4 depicts the *RIJID* indices of cryptographic programs and their loop size in number of instructions when imposed on *A-RIJID*. RSA, Blowfish and Rijndael provide *RIJID* indices of 0.9980, 0.9622 and 0.9495, while TripleDES and SHA, provide 0.7040 and 0.7096 respectively. Due to the enormous amounts of time taken for power simulations, we only considered these five benchmarks.

	Loop Size	RIJID index
TripleDES	14	0.7040
Blowfish	37	0.9622
Rijndael	109	0.9495
SHA	18	0.7096
RSA	36	0.9980

Table 4: RIJID index using A-RIJID

8.4 DPA and A-RIJID

Figure 5 shows the DPA performed on TripleDES (based on the technique explained in [5]), an attempt to prove that our method prevents DPA. TripleDES was chosen because it provides the lowest RIJID index, thus the most vulnerable amongst the programs shown above. Plots are provided for each selection bit where the last SBOX lookup of the 16th round is chosen as the attacking point. Figure 5(a) shows DPA plots on TripleDES without the *A-RIJID* implementation and Figure 5(b) shows using *A-RIJID* applied.

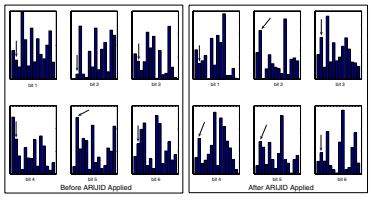

Figure 5: DPA before and after A-RIJID

Carefully chosen keys (just 14 out of the possible 2^{64}, including the correct one were chosen to demonstrate within the available space) were guessed such that DPA can be demonstrated using fewer power samples. Note that in a real implementation the adversary needs to consider all possible key guesses.

As Figure 5(a) depicts, the key is successfully predicted (place of the correct key is pointed to by arrows in each plot) when using $bit5$ (the second plot in the second row of Figure 5(a)) of the 6 selection bits used for SBOX lookup. All the other bits except $bit5$ (Note that the bits are counted from the least significant bit - right to left) fail to give a peak at the correct key as shown in Figure 5.

The key cannot be predicted using any of the bits after *A-RIJID* is applied as shown in Figure 5(b), where no peaks appear on the correct key. This analysis demonstrates that DPA will not work even if the adversary manages to eliminate the injected instructions, identifying the proper places in the power sequence for the analysis.

9. CONCLUSION

This paper proposes a random instruction injection technique (*A-RIJID*) using dynamic signature detection to prevent side channel attacks. Two different signatures are defined to identify critical blocks (such as encryption blocks) in a cryptographic program. Random number of instructions at random places are injected during the runtime of the processor, based on the detected signatures, to scramble the power wave so that adversaries cannot extract any useful information by observing the power wave leakage from the processor.

Our *A-RIJID* processor consumes an area overhead of 1.2%, an average runtime overhead of 25% and an average energy overhead of 28.5%. As far as we are aware this is the most cost effective solution, with no manual intervention needed. The downside of this approach is the small overhead, non-cryptographic programs can occasionally encounter.

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 methodology to work with superscalar processors and VLIW processors.

10. REFERENCES

[1] Sourcebank. url: http://archive.devx.com/sourcebank.
[2] The PEAS Team. ASIP Meister, 2002. Available at: http://www.eda-meister.org/asipmeister.
[3] 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.
[4] M. Barbosa and D. Page. On the automatic construction of indistinguishable operations. In *Cryptography And Coding*, pages 233–247. Springer-Verlag LNCS 3796, November 2005.
[5] E. Brier, C. Clavier, and F. Olivier. Correlation power analysis with a leakage model. In *CHES*, pages 16–29, 2004.
[6] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX*, August 2003.
[7] 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.
[8] 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.
[9] J.-S. Coron and L. Goubin. On boolean and arithmetic masking against differential power analysis. In *Ches '00*, pages 231–237, London, UK, 2000.
[10] J. Daemen and V. Rijmen. Resistance against implementation attacks: a comparative study of the AES proposals, 1999.
[11] B. S. David, C. Lap-Wai, and M. C. William. Cryptographic architecture with random instruction masking to thwart differential power analysis. *U.S. Patent 20050271202*, 2005.
[12] C. Gebotys. A Table Masking Countermeasure for Low-Energy Secure Embedded Systems. *IEEE Trans. on VLSI*, 14(7):740–753, 2006.
[13] 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.
[14] A. Gordon-Ross and F. Vahid. Frequent loop detection using efficient non-intrusive on-chip hardware. In *Cases '03*, pages 117–124, New York, NY, USA, 2003.
[15] L. Goubin and J. Patarin. Des and differential power analysis (the "duplication" method). In *Ches '99*, pages 158–172, London, UK, 1999. Springer-Verlag.
[16] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Wwc '01*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.
[17] J. Irwin, D. Page, and N. P. Smart. Instruction stream mutation for non-deterministic processors. In *Asap '02*, page 286, Washington, DC, USA, 2002.
[18] A. Janapsatya, A. Ignjatovic, and S. Parameswaran. Exploiting statistical information for implementation of instruction scratch memory in embedded system. *IEEE Trans. on VLSI*, 14(8):816–829, 2006.
[19] S. Mangard. A Simple Power-Analysis (SPA) Attack on Implementations of the AES Key Expansion. In P. J. Lee and C. H. Lim, editors, *icisc 2002*, volume 2587 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2003.
[20] D. May, H. L. Muller, and N. P. Smart. Non-deterministic processors. In *Acisp '01*, pages 115–129, London, UK, 2001. Springer-Verlag.
[21] 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.
[22] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–589, 2001.
[23] 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.
[24] R. Muresan and C. H. Gebotys. Current flattening in software and hardware for security applications. In *CODES+ISSS*, pages 218–223, 2004.
[25] E. Oswald and M. Aigner. Randomized addition-subtraction chains as a countermeasure against power attacks. In *CHES '01:*, pages 39–50, London, UK, 2001.
[26] 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.
[27] Paul Bourke. Cross Correlation: AutoCorrelation – 2D Pattern Identification. url:http://astronomy.swin.edu.au/ pbourke/other/correlate/index.html, 1996.
[28] J. J. Paul Kocher and B. Jun. Differential Power Analysis. 1998. First article on DPA.
[29] J.-J. Quisquater and D. Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *E-smart*, pages 200–210, 2001.
[30] 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.
[31] 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.
[32] 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.
[33] 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.
[34] J. Waddle and D. Wagner. Towards efficient second-order power analysis. In *CHES*, pages 1–15, 2004.