Randomized Instruction Injection to Counter Power Analysis Attacks

JUDE A. AMBROSE, University of New South Wales
ROSHAN G. RAGEL, University of Peradeniya
SRI PARAMESWARAN, University of New South Wales

Side-channel attacks in general and power analysis attacks in particular are becoming a major security concern in embedded systems. Countermeasures proposed against power analysis attacks are data and table masking, current flattening, dummy instruction insertion and bit-flips balancing. All these techniques are either susceptible to multi-order power analysis attack, not sufficiently generic to cover all encryption algorithms, or burden the system with high area, run-time or energy cost. In this article, we propose a randomized instruction injection technique (*RIJID*) that overcomes the pitfalls of previous countermeasures. *RIJID* scrambles the power profile of a cryptographic application by injecting random instructions at random points of execution and therefore protects the system against power analysis attacks. Two different ways of triggering the instruction injection are also presented: (1) *softRIJID*, a hardware/software approach, where special instructions are used in the code for triggering the injection at runtime; and (2) *autoRIJID*, a hardware approach, where the code injection is triggered by the processor itself via detecting signatures of encryption routines at runtime. A novel signature detection technique is also introduced for identifying encryption routines within application programs at runtime. Further, a simple obfuscation metric (*RIJIDindex*) based on cross-correlation that measures the scrambling provided by any code injection technique is introduced, which coarsely indicates the level of scrambling achieved. Our processor models cost 1.9% additional area in the hardware/software approach and 1.2% in the hardware approach for a *RISC* based processor, and costs on average 29.8% in runtime and 27.1% in energy for the former and 25.0% in runtime and 28.5% in energy for the later, for industry standard cryptographic applications.

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

General Terms: Security, Measurement

Additional Key Words and Phrases: Side channel attack, random instruction injection, pattern matching, cross correlation, power analysis attack, methodology

ACM Reference Format:
Ambrose, J. A., Ragel, R. G., and Parameswaran, S. 2012. Randomized instruction injection to counter power analysis attacks. ACM Trans. Embed. Comput. Syst. 11, 3, Article 69 (September 2012), 28 pages.
DOI = 10.1145/2345770.2345782 http://doi.acm.org/10.1145/2345770.2345782

1. INTRODUCTION

Over the past decade, side-channel attacks have become a serious concern in embedded system security. Adversaries measure side channels such as the power usage [Mangard 2003], the processing time [Brumley and Boneh 2003] or the electromagnetic emission [Quisquater and Samyde 2001] of an application, so that they could correlate these

Authors' addresses: A. Ambrose and S. Parameswaran, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia; R. G. Ragel, Department of Computer Engineering, University of Peradeniya, Sri Lanka; email: roshanr@pdn.ac.lk.

DOI 10.1145/2345770.2345782 http://doi.acm.org/10.1145/2345770.2345782

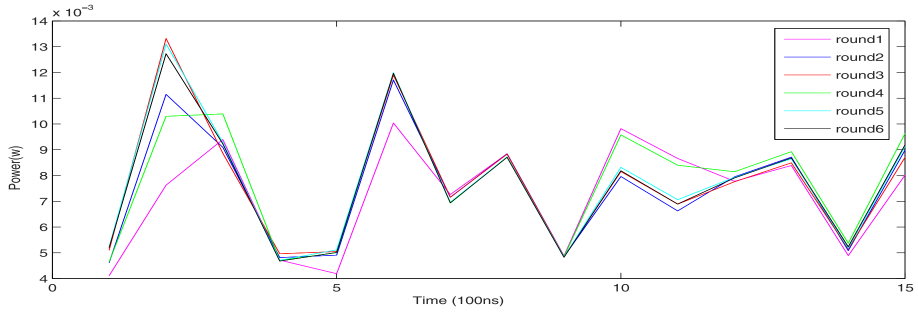


Fig. 1. TripleDES rounds.

external manifestations with internal computations. These properties are used to obtain critical information, such as the secret key of a secure application. Analyzing the power profile of a cryptographic algorithm, also known as the *power analysis*, has been the most effective technique to extract the secret key using side channel attacks [Mangard 2003; Messerges et al. 2002; Ors et al. 2004].

There are two types of power analysis attacks used and they are: (1) simple power analysis (SPA) [Mangard 2003]; and (2) differential power analysis (DPA) [Kocher et al. 1999]. SPA involve the identification of computations (and therefore the instructions) used in a system by analyzing the power profile observed. 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 such as AES and TripleDES.

In a power analysis attack, a typical attacker would observe the patterns occurring in the power waveform as depicted in Figure 1. Such a pattern is referred to as a *template* from this point onwards. Figure 1 depicts six TripleDES *sbox* (a fixed or a dynamic mapping table that returns values for cipher text, based on the plain text and decides the strength of the encryption algorithm) lookup rounds. All power waves for the shown rounds appear similar in Figure 1. By correlating power values of identified segments with guessed secret keys, the adversary will identify the correct secret key using fewer power samples than if brute force were to be applied.

The secret key of a system is usually revealed by analyzing the differences in magnitudes of power values between differing rounds in a cryptographic algorithm [Kocher et al. 1999; Mangard 2003; Ravi et al. 2004]. For example, in Figure 1, multiple peaks at approximately 250ns show differing parameters used in an XOR instruction. SPA attacks will only be successful when instructions executed have an obvious and simple relationship with the secret key [Blake et al. 2005; Mayer-Sommer 2000; Messerges et al. 1999; Novak 2002]. DPA attacks are much more powerful than SPA attacks, as statistical analysis is used on the observed power wave to find secret keys [Kocher et al. 1999; Oswald et al. 2006]. However, 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 analysis attack. This is the concept used in this article to protect embedded systems against power analysis attacks.

In this article, we introduce a *Randomized Instruction inJectIon methoD*, called *RIJID*, that scrambles the power profiles of cryptographic program. *RIJID* will stop an adversary from identifying specific segments (such as encryption rounds) within the entire power profile of an application. It is worth noting that our method injects *real* instructions at random places as opposed to dummy instructions like *NO-OP* at fixed places [Akkar et al. 2000]. The two major steps followed by *RIJID* to inject instructions are: (1) identifying the *critical routines* of a cryptographic algorithm and (2) injecting random instruction during the execution of these critical routines. While we achieve the second step via hardware modification, we explore two different ways to perform the first one: (1) via a software instrumentation we call *softRIJID*, where the programmer is expected to identify critical routines and tag (with special instructions) them statically [Ambrose et al. 2007a], and (2) via a hardware signature detector we call *autoRIJID* (for automatic *RIJID*), where the processor designer is expected to define the signatures of critical routines that are expected to be identified at runtime by the special hardware [Ambrose et al. 2007b]. Even though *autoRIJID* releases the programmer's responsibility of identifying and tagging critical routines, *softRIJID* can be applied to any program with a minimal hardware modification compared to *autoRIJID*.

In addition to *RIJID*, we have also proposed a new metric to measure the degree of obfuscation performed to an application. This metric, called the *RIJIDindex*, uses cross-correlation to give us a measure of obfuscation. This metric allows us to quickly find the level of obfuscation needed to counter power analysis attack. However, to be absolutely certain, one must measure power and try to perform power analysis attack, which would take a very long time. *RIJIDindex* assess the vulnerability of the power profile for detecting critical patterns, which are used by the adversaries for power analysis. There exist approaches [Macé et al. 2007; Regazzoni et al. 2009] in the literature to assess the power leakage for analysis after the critical power patterns are extracted from the power trace. We see *RIJIDindex* as a complementary metric to what is proposed in the literature that can be used for initial assessment.

This article critically compares the approaches presented in Ambrose et al. [2007a], the software/hardware technique, and Ambrose et al. [2007b], the hardware-based technique. In addition, we have added the following: (1) detailed explanation with additional experimental results are provided on the signature detection system; and (2) the use of *RIJIDindex* to analyze our solution is explained in detail with additional information.

Motivation. The technique of obfuscating power signals to confuse the adversary covers methods from constant path execution, current flattening, masking and dummy instruction insertion. All these are software methods, requiring a high degree of manual intervention. Additionally, these methods increase the software footprint of the application substantially, as well as consume copious amount of additional power. On the other hand, the circuit-based methods include constant signal suppression and balanced logic, both consume substantial amount of area and energy. The work presented in this article, in contrast to previous obfuscation methods, modifies the processor itself to inject random instructions (not just *NO-OPs*, but a selected set of instructions, with randomized operands to change hamming weights). In the rest of this subsection, we give the reason for not choosing *NO-OPs* for our instruction injection.

Figure 2 depicts the power sequences when injecting dummy and real instructions to a code segment. Figure 2(a) and Figure 2(b) depict six templates (circled and numbered) without and with dummy instruction injected. As in Figure 2(b), the injections are distinguishable in the power wave as significant troughs, while the rest of the patterns remain the same as the original. Simple time shifting [Clavier et al. 2000] can be deployed to eliminate the effect of dummy instructions to extract the original power profile. Since dummy instructions do not perform operations in all pipeline stages (except the fetch stage), such instructions will cause only minor changes on the power profile of neighboring instructions.

Figure 2(c) shows the power profile of the code segment when real instructions are injected using *RIJID*. There are no patterns or significant troughs prevail in the power

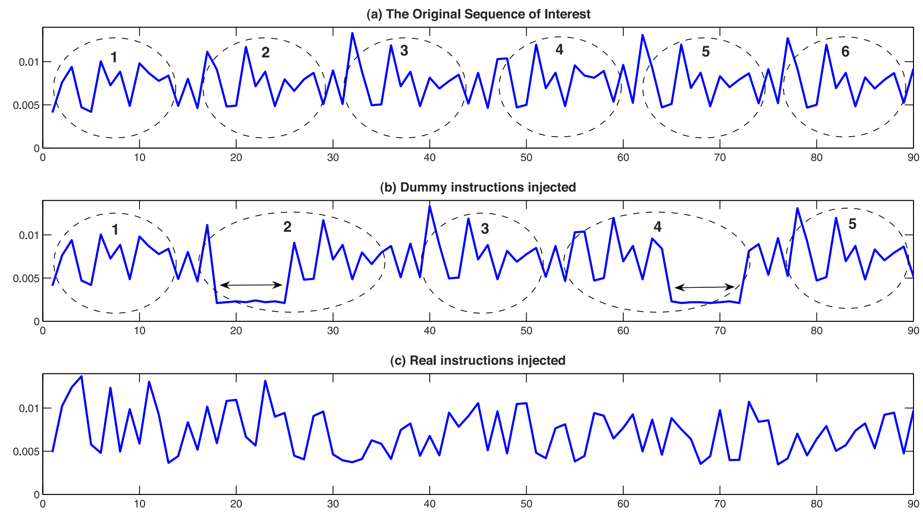


Fig. 2. Dummy vs real instruction injection.

wave, hence the insertions cannot be spotted. *RIJID* uses random operands and random registers (note that the usage of random registers and operands lead to differing amounts of bit-flips, further obfuscating the power profile). Since these are real instructions (such as *AND*, *OR* etc.), they will also change the power dissipated by neighboring instructions due to the pipeline in the processor. Time shifting as in the case of Figure 2(b) cannot be applied to predict the original sequence, because the obfuscated sequence (shown in Figure 2(c)) appears as some other *random* sequence.

Article Organization. The rest of the article is organized as follows. Section 2 investigates previous research on countermeasures against power analysis attacks. Key hardware/software design techniques of *softRIJID* is presented in Section 3 and of *autoRIJID* is presented in Section 4. Design flows of *softRIJID* and *autoRIJID* are explained in Section 5. The formulation of *RIJIDindex* is presented in Section 6. Section 7 explains the experimental setup used for measurement and analysis. Results are shown in Section 8. Finally, the article is concluded in Section 9.

2. RELATED WORK

Kocher et al. [1999] employed side channel attacks on embedded systems using SPA and DPA techniques and suggested several solutions to counteract these attacks (such as choosing operations that leak little or no information, adding noise and physically shielding the embedded system). Countermeasures proposed by researchers to prevent power analysis attacks could be divided into a number of categories such as masking, nondeterministic processing, current/power flattening, balancing and circuitry level solutions.

Kocher et al. [1999] 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 [Messerges et al. 2002]) using random arbitrary values or functions combining the actual data, is a well-known countermeasure. In masking, a random value is used with the actual secure computation to confuse the adversary such that wrong data values are predicted [Chari et al. 1999; Coron and Goubin 2000; Standaert et al. 2005]. For example, in the AES cryptographic algorithm the intermediate result after each round, which is a vulnerable place, can be masked by having an additional computation with a random value [Trichina et al. 2003]. The duplication method [Goubin and Patarin 1999] and table masking [Gebotys 2006] 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 [Saputra et al. 2003]. Coron explains a randomization masking technique on Elliptic Curve Crypto-systems (ECC) to counteract DPAs, where a random number is used to compute the points on the curve for encryption [Coron 1999]. Constant execution path or designing a piece of code to always yield the same result [Barbosa and Page 2005; Kocher et al. 1999] is another masking technique, where the adversary will not be able to predict the computations happening inside the system.

A nondeterministic processor [May et al. 2001a], which uses random selection circuitry, is used to perform random issuing of independent code segments during runtime. In this technique the adversary cannot predict the instructions if they are executed out-of-order. Irwin et al. [2002] presented a software and hardware technique for nondeterministic processors that uses an additional pipeline stage that performs random operations without modifying the effective data. In May et al. [2001b], a random register renaming technique is proposed for the nondeterministic processor designed in May et al. [2001a], which uses a logic circuit to rename the internal registers randomly, depending upon the availability to hide information leaks from secret key computations.

Expanding the bus size to cause equal bit flips during operations is another technique to prevent power analysis attacks [Saputra et al. 2003]. Such balancing of bit flips has been further looked at in Hwang et al. [2006] and May et al. [2001a].

A current flattening technique is proposed in Muresan and Gebotys [2004] to flatten the power wave of a processor. However, their technique flattens the current within basic block of execution. The secure coprocessor [Tiri et al. 2005], which is designed for AES-based biometric applications, uses a constant power dissipating logic for any bit transitions. This coprocessor costs 3X in area and 4X in power. A signal suppression technique is proposed in Ratanpal et al. [2004], 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 (*NO-OPs*) could be a solution to protect systems from power analysis attack [Akkar et al. 2000; Kocher et al. 1999]. However, to our knowledge none of them have been implemented. An application specific dummy operation insertion technique is proposed for ECC systems to create a constant execution path in Barbosa and Page [2005] and Gebotys and Gebotys [2003]. Clavier et al. [2000] proposed an improved DPA attack called Sliding Window DPA (SW-DPA), to bypass the dummy instruction insertion technique. Daemen and Rijmen [1999] state that inserting dummy instructions is not an appropriate solution where each instruction has its own power profile, and synchronization is possible. However, for pipelined systems such as the one proposed in this article, each instruction's power profile affects the power profile of its neighboring instructions due to the overlapping pipeline stages. Such overlapping instructions mitigate what was stated by Daemen and Rijmen regarding single instructions. In this article, we prove that this argument is valid.

In general, data masking techniques have been vulnerable to second-order DPA attacks [Joye et al. 2005; Oswald et al. 2006; Waddle and Wagner 2004]. Table masking methods [Gebotys 2006; Goubin and Patarin 1999] are algorithm specific approaches (works for algorithms that use an *sbox*) and they successfully prevent DPA. However, masking techniques require a higher 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 [Barbosa and Page 2005; Gebotys and Gebotys 2003] are application specific and needs significant human intervention. Nondeterministic processors [May et al. 2001a] are not feasible in highly dependent software code that cannot be executed *out-of-order*. The techniques proposed using nondeterministic processors [Irwin et al. 2002; May et al. 2001b] have complex circuitries but do not have their overheads reported, as none have been implemented to the best of our knowledge. The circuitry level solutions [Ratanpal et al. 2004; Tiri et al. 2005] cost significant area and energy overheads. The signal suppression technique [Ratanpal et al. 2004] does not completely prevent DPA, but tries to make the attack more difficult. Balancing bit flips is also a costly solution that increases the size of the components used (e.g., in Saputra et al. [2003], bus size is doubled to balance the bit flips). The current flattening technique, which is considered the most appropriate countermeasure for power analysis based side channel attacks increases execution time by up to 75%, and flattens locally, based upon basic blocks.

As opposed to the pitfalls from previous methods, *RIJID* provides a generalized solution with little human intervention compared to the masking methods [Chari et al. 1999; Coron and Goubin 2000; Gebotys 2006; Goubin and Patarin 1999; Standaert et al. 2005], allowing the processor to take care of masking. On a processor with PISA instruction-set (as implemented in *SimepleScalar*™), the additional area cost for *RIJID* are just 1.98% for *softRIJID* and 1.20% for *autoRIJID*. The average energy and runtime cost are 27.1%, 29.8% for *softRIJID* and 28.5%, 25.0% for *autoRIJID* for industry standard embedded system benchmarks. *RIJID* confuses the adversary as opposed to flattening the current[Muresan and Gebotys 2004]. It can be applied to any vulnerable segment (and is not algorithm dependent like masking techniques [Gebotys 2006; Standaert et al. 2005]). Dummy instruction insertions can be eliminated using simple time shifting [Clavier et al. 2000], whereas *RIJID* injects real instructions at random places a random number of times. Hence the adversary will observe different power profiles on different tries.

The signature detection presented for *autoRIJID* is a novel idea for encryption instruction sequence detection. Several researchers have implemented methodologies to find the frequently executed loops using signatures based on jump instructions [Gordon-Ross and Vahid 2003; Merten et al. 2001] for HW/SW codesign methods. Thus far, there has been no implementation of signatures to identify critical segments (such as sequence of instructions performing encryption) in a cryptographic program. Several secure operations (like random instruction injection proposed in this article) could be automated by predefining such signatures inside the processor. Hence we present a novel signature based detection system to identify critical segments for the instruction injection.

2.1. Contributions

The contributions of this article are as follows:

(1) a randomized instruction injection technique that inserts random *real* instructions at random places to scramble the power profile is presented with two different implementations: *softRIJID* and *autoRIJID*;
(2) a novel signature detection technique to identify critical routines of a cryptographic application; and
(3) a simple obfuscation metric is introduced (called *RIJIDindex*) based on cross-correlation that measures the scrambling provided by any code injection technique, which coarsely indicates the level of scrambling achieved.

2.2. Limitations

The limitations of *RIJID* are:

(1) *RIJID* is proposed as a design-time technique and therefore it needs hardware changes;
(2) *softRIJID* needs compiler support;
(3) new signatures need to be added for *autoRIJID* when an entirely different encryption algorithm of different nature is introduced;
(4) *RIJIDindex* is a relative measure that can be used to compare one power profile against the other that uses code injection techniques to scramble. It is apparent that we cannot compare a wave produced by, for example, a hardware balancing technique and a wave produced by the injection technique. The main reason is that *RIJIDindex*considers only the code injection scenario at the moment. Though this limitation can be overcome by an improved version to compare different techniques, it is beyond the scope of this article. However, as stated in Contribution 3, RIJIDindex can be used to compare waves that utilize any CODE INJECTION techniques to scramble the critical patterns in the power profile.
(5) we assume that our system is self contained with memory on chip; and
(6) theoretically, the randomness introduced by *RIJID* could be removed from the power profile. However, the attack would require a huge number of power traces, making power analysis attacks impractical; and
(7) the current version supports scalar architectures as our target is embedded processors. The design needs to be improved to support processors that incorporate advanced super-scalar features. We propose this as future work.
(8) we assume that our system is not vulnerable to any other software/hardware attacks, such as code injection. Further we assume a trusted OS.

3. SOFTRIJID FRAMEWORK

In this section, we present an overview of the software (for tagging the code segments at compile time) and hardware (for injecting random instructions at runtime) architecture of *softRIJID* framework.

3.1. softRIJID: Software Instrumentation

The software instrumentation for tagging critical code segments for *softRIJID* is performed at compile time. Figure 3 depicts a sequence of instructions performing encryption from a cryptographic application that 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 instruction sequence as shown in the figure to indicate the starting and ending points for inserting random instructions. When the processor fetches a *SET-FLAG* instruction, it starts generating random instructions and stops when it fetches the *RESET-FLAG* instruction.

Multiple occurrences of a set of instructions, which causes a repeated template is a potential critical instruction sequence that performs encryption. Recent SPA and DPA attacks reveal that the sequence of instructions performing encryption that has *sbox* rounds is the most critical part [Oswald et al. 2006], which is selected for random instruction injection in our technique. In *softRIJID* the programmers decide the routines upon which to apply flag instructions. However, this instrumentation can be automated for any given code, by having a parser that identifies encryption loops or by using a more sophisticated runtime technique as presented in the next section under *autoRIJID*.

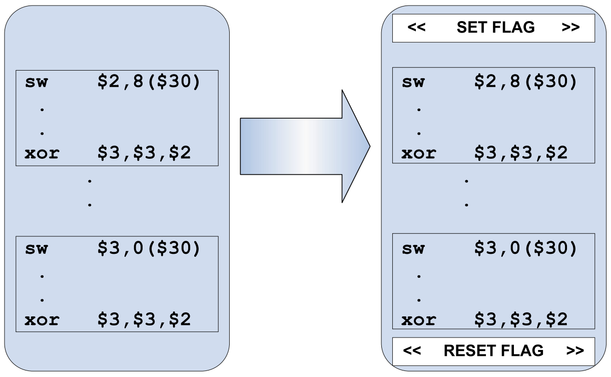


Fig. 3. Software instrumentation at compile time.

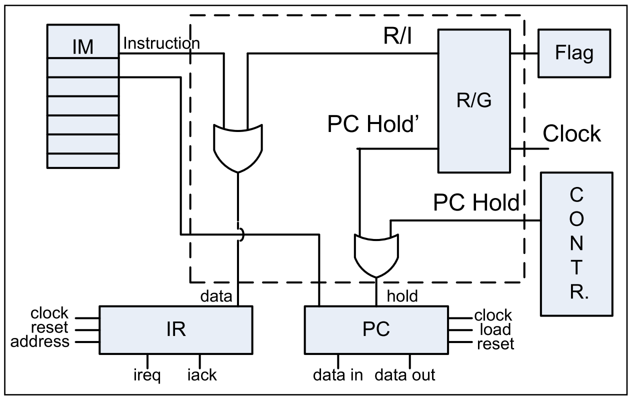


Fig. 4. Random instruction injection.

3.2. *softRIJID*: Hardware Architecture

The block diagram in Figure 4 is the hardware architecture of *softRIJID* framework. When the processor fetches a *SET-FLAG* instruction, a special register (*Flag*) is set. The dotted box in Figure 4 depicts the hardware components added for *RIJID* framework. When *Flag* is set, the random generator component (R/G) sends a hold signal (*PC Hold'*) to the program counter (PC) and starts generating random instructions. The *PC Hold'* signal from R/G and the *PC Hold* signal from the controller (*CONTR.*) are multiplexed before they are connected to the *PC*.

Random instructions generated by R/G are limited by a pair of boundary values (N,D), which are set by the *SET-FLAG* instruction. This pair, called the *injection pair* represents: (1) the maximum number of random instructions to be injected between two regular instructions when the flag is set (*N*); and (2) the maximum number of regular instructions to be skipped before each injection (*D*). Figure 5 depicts four possible instruction sequences when an injection pair of (3,4) is set ('x' in the figure denotes an injected instruction).

PC Hold' signal from R/G is switched between on and off for random intervals with an upper-bound that is defined by the *injection pair*. The generated random instruction (R/I) is multiplexed with the data bus (*Instruction*) of the instruction memory (*IM*). The multiplexers are selected by R/G based on the instruction injection. When *PC Hold'* is high, the random instructions (R/I) generated by R/G are sent to the data

Inst. Seq.	**ABCAABCDEFGAABBCDEF**
	ABxxCAABxxCDExFGxxxAABBxxCDExF
	ABCAxxxABCxDEFxxGxAxxABBxxCDxxxEF
	ABCAxABCxDxxEFxxGAAxxxBxBCxxDEF
	AxxBCxxAAxBCxDExFxxGAxxxABxxBCxDxEF

Fig. 5. Instruction sequence possibilities for injection pair (3,4).

port of the instruction register (*IR*). During hold, instruction that is pointed to by *PC* is also refetched from the instruction memory, but, is not written into the *IR*.

Given that the execution of an instruction will generally affect the state of a processor, injecting any random instruction will overwrite or amend effective data values. Therefore, only a limited set of instructions is selected such that the execution of the set will not change the state of the processor. Three different schemes were evaluated to be the candidate for the limited set: (1) instructions with only *zero* registers; (2) instructions with *zero* and a fixed register; and (3) instructions with *zero* and a randomly selected register. The insertions from (1) and (2) are identifiable on the power wave due to the lower amount of power variation we get from these schemes. The third scheme was chosen as the most appropriate technique for *RIJID* as it causes higher power variation due to bit flips in registers as expected [May et al. 2001b]. In the third scheme, a random register is used for computation with the *zero* register and the result is written onto the same random register. Since R/G generates random numbers using a simple Linear Feedback Shift Register (LFSR), it is called a pseudo random number generator (PRNG). As an example, the PRNG is used to randomly choose *ADD* instruction (from a pool of instructions: XOR, OR, ORI, ADD, and ADDI), to add the value in a random register (say *R10*) with the zero register. Other instructions could be added in a more sophisticated processor. The status register is also saved into another register before every instruction injection and restored afterwards, since the status register can be modified by the execution of the injected instructions. As the aim of this article is to show that the proposed *RIJID*approach works and True RNGs (TRNG) are slow in simulation [Boubekeur and Schlick 2007], we have used a simple PRNG instead of a TRNG. A TRNG will increase the area overhead of the *RIJID*technique.

4. AUTORIJID FRAMEWORK

In this section, we present an overview of the *autoRIJID* framework. *autoRIJID* uses a signature detection technique, known as concomitance, to automatically identify critical code segments at runtime. Later in this section, we present the extensions we made to the *softRIJID* architecture that automatically starts injecting instructions when signatures are detected and stops when they expire.

4.1. Concomitance

Concomitance is a measure of how frequently and closer instructions are executed using a trace. We use this measure to define signatures for encryption routines in cryptographic program. The *concomitance metric* presented in Janapsatya et al. [2006] is used in this article with modification to support our analysis.

As shown in Equation (1), self-concomitance $\sigma(b, T)$ of an instruction is a measure of how clustered consecutive executions of the instruction b are, in the execution trace (T). The distance between two consecutive executions $e(b)$ and $e'(b)$ of an instruction b is referred to as d, which is the number of instructions executed in between $e(b)$ and $e'(b)$ including b. A weight function called W is used to find the concomitance using d as presented in Equation (2). W is used to give a decreasing significance to the two

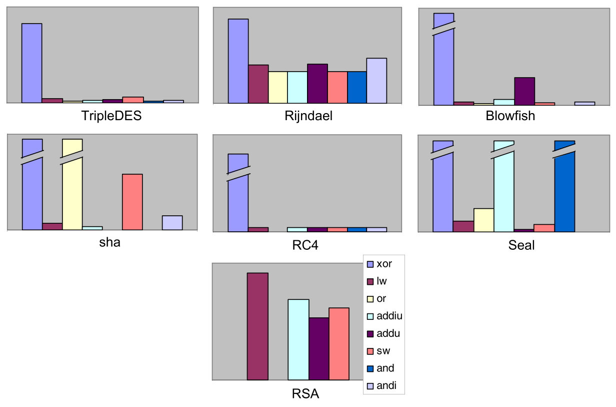


Fig. 6. Self-concomitance.

consecutive executions of the same instruction that are further apart in the sense of the above notion of distance. Thus, it is a nonnegative real function that is decreasing such that, if u, v are real numbers and $u \leq v$, then $W(u) \geq W(v)$.

$$\sigma(b\,, T) = \sum_{e(b)\epsilon T} W(d[e(b\,), e'(b\,)]) \tag{1}$$

$$W(d[e(b\,), e'(b\,)]) = 1/d[e(b\,), e'(b\,)] \tag{2}$$

The self-concomitance is computed as a sum of all weights on instruction b in the execution trace (T) as shown in Equation (1). Figure 6 depicts the self-concomitance analysis on instructions in cryptographic programs.

The plotted values in Figure 6 are the ratio ($\Upsilon(b\,, T)$) of self-concomitances with ($\sigma(b\,, T)_{w.e}$) and without ($\sigma(b\,, T)_{w.o.e}$) the encryption routines within cryptographic programs, computed using Equation (3). The self-concomitance ratio ($\Upsilon(b\,, T)$) for the cryptographic programs in Figure 6 are much higher for *xor* instructions compared to other instructions (except for RSA). Even though some instructions provide a higher value similar to *xor* such as *or* in sha and *addiu*, *and* in seal, *xor* gives a higher value on all the tested programs, except RSA. This denotes that *xor* instructions are mostly used within the sequence of instructions performing encryption of cryptographic programs, giving a higher $\Upsilon(xor, T)$.

$$\Upsilon(b\,, T) = \sigma(b\,, T)_{w.e}/\sigma(b\,, T)_{w.o.e} \tag{3}$$

A threshold-concomitance is introduced to find the most appropriate distance between *xor* instructions such that all the *xor* instructions inside the sequence of instructions performing encryption can be included for concomitance. This is to make sure that none of the encryption part is left out without balancing. The weight function W is modified to W_t as shown in Equation (4) to find the threshold-concomitance by changing different c values. When distance d is greater than c, the weight added is zero in the summation. The self-concomitance values are computed for each c using Equation (1), substituting weight W_t. This measure allows us to predict the distance that we have to move to cover all the consecutive *xor* instructions inside the encryption instruction sequence. Figure 7 depicts the threshold-concomitance analysis for *xor* instruction in cryptographic programs, where self-concomitance is computed, incrementing c by one starting from one until the value gets stabilized (note that the

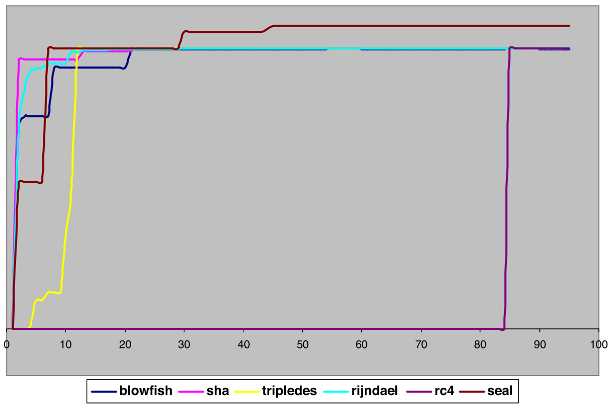


Fig. 7. Threshold-concomitance for XOR.

values are scaled to show the variation). The distance where the self-concomitance stabilizes is referred as threshold-concomitance.

$$W_t(d[e(b), e'(b)]) = \begin{cases} 0 & if d[e(b), e'(b)]\ c, \text{where c} \in \mathbb{R}, \quad (4) \\ 1/d[e(b), e'(b)] & if d[e(b), e'(b)] \le c. \quad (5) \end{cases}$$

According to Figure 7, the maximum threshold-concomitance is for rc4 with 85, with 55 for blowfish, 45 for seal and 30, 21 and 12 for rijndael, sha and tripleDES respectively. Therefore, 85 is considered as the appropriate choice for threshold-concomitance in all the analyzed cryptographic programs (except RSA), where all the *xor* instructions inside the encryption routines are included.

As presented in Figure 6, the RSA cryptographic program has zero $\Upsilon(xor, T)$ for *xor* instruction, which is the appropriate choice for signature in other cryptographic programs. A concomitance analysis is used to find any possible instruction combination to differentiate the encryption instruction sequence within RSA from the rest of the program. Concomitance predicts how tightly interleaved the executions of two instructions are, in the execution trace. Equation (6) depicts the concomitance $\tau(a, b, T)$ of instructions a and b in the execution trace T. There should be at least one b instruction execution $e(b)$ between two a instruction executions, $e(a)$ and $e'(a)$: we denote this by $b \in [e(a), e'(a)]$. The weight function W is computed from Equation (2).

$$\tau(a, b, T) = \sum_{b \in [e(a), e'(a)],\ e(a) \in T} W(d[e(a), e'(a)]) + \sum_{a \in [e(b), e'(b)],\ e(b) \in T} W(d[e(b), e'(b)]) \quad (6)$$

The concomitance analysis performed on RSA program is presented in Figure 8. The concomitance ratios $(\Upsilon(a, b, T)$ as shown in Equation (7) are plotted, computing the concomitance with the sequence of instructions performing encryption $(\tau(a, b, T)_{w.e})$ and without the encryption instruction sequence $(\tau(a, b, T)w.o.e)$. As Figure 8 shows, the maximum ratio is produced when using *mult* followed by *div*. All the other possible instruction combinations except the plotted ones gave zero ratio $(\Upsilon(a, b, T)$=0). From this analysis, we can conclude that $mult - div$ instruction combination is the appropriate choice to detect the encryption instruction sequence within RSA (where removal

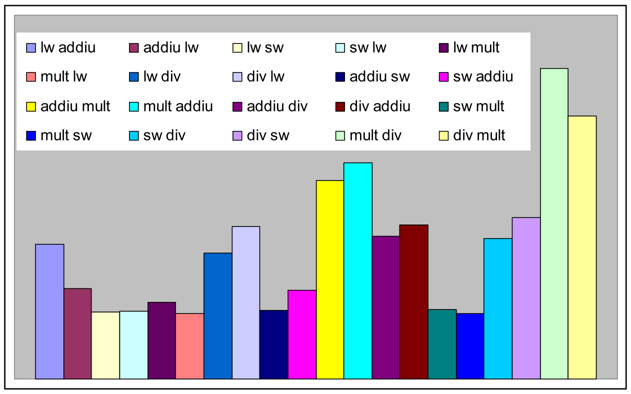


Fig. 8. Concomitance on RSA.

of the encryption instruction sequence causes higher ratio on $mult - div$, compared to other instruction combinations).

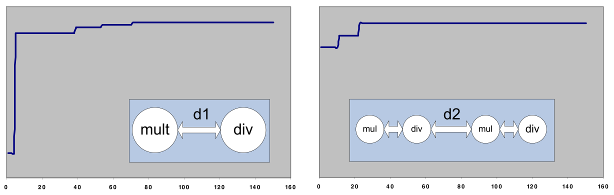
$$\Upsilon(a, b, T) = \tau(a, b, T)_{w.e} / \tau(a, b, T)_{w.o.e} \tag{7}$$

$$\begin{aligned} \tau(a, b, T) &= \sum_{e(a),e(b)\epsilon T} W_{rt}(d[e(a), e(b)]) + \sum_{e(b),e'(a)\epsilon T} W_{rt}(d[e(b), e'(a)]) + \\ &\qquad \sum_{e'(a),e'(b)\epsilon T} W_{rt}(d[e'(a), e'(b)]) \\ &= \sum_{e(a),e(b)\epsilon T} W_{rt}(d1) + \sum_{e(b),e'(a)\epsilon T} W_{rt}(d2) + \sum_{e'(a),e'(b)\epsilon T} W_{rt}(d3) \end{aligned} \tag{8}$$

$$\begin{aligned} W_{rt}(d1) &= \begin{cases} 0 & \text{if } d1 > c, \text{where c } \epsilon\ \mathbb{R}, \\ 1/d1 & \text{if } d1 \le c \end{cases} \\ W_{rt}(d3) &= \begin{cases} 0 & \text{if } d3 > c, \text{where c } \epsilon\ \mathbb{R}, \\ 1/d3 & \text{if } d3 \le c \end{cases} \\ W_{rt}(d2) &= 1/d2. \end{aligned} \tag{9}$$

Threshold analysis is performed on RSA to decide two main distances: (1) the most appropriate distance between *mult* and *div* instructions; and (2) the most appropriate distance between $mult - div$ segments, for the signature definition. The concomitance equation shown in Equation (6) is slightly modified to Equation (8) for the threshold-concomitance analysis on RSA. Distances $d1$ and $d3$ separate *mult* then *div* and distance $d2$ separates two consecutive $mult - div$ segments.

Figure 9(a) depicts the variation of the concomitance metric, when changing the distance between *mult* and *div* instructions (referred as $d1$ in Figure 9(a)) using Equation (8) and Equation (9) for RSA. The value c is incremented by one starting from one, until the concomitance metric stabilizes. When threshold value c is less than the distances, corresponding weights are assigned to zeros as shown in Equation (9). The concomitance metric stabilizes at distances 5, 40, 60, and 70 as shown in Figure 9(a). To capture only the sequence of instructions performing encryption using *mult*,*div* instruction combination, the threshold-concomitance of *five* is considered as



(a) threshold-concomitance: distance between mult – div instructions (b) threshold-concomitance: distance between subsequent mult-div combinations

Fig. 9. Threshold-concomitance for RSA.

Table I. Instruction hits on BenchMarks

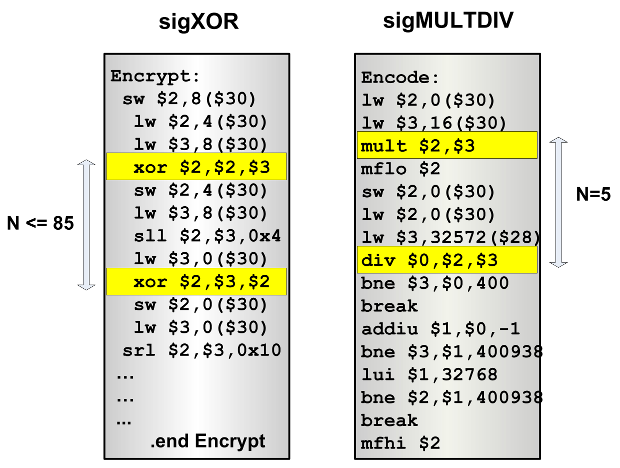
	Programs	**xor hits**	**m hits**	**Inst.**	**% X**	**% M**
G	Dijsktra	686	0	975333	0.00	0.00
E	JPEG	3	0	9167386	0.00	0.00
N	FFT	449	0	732776	0.06	0.00
E	QSORT	23	0	22684	0.10	0.00
R	BasicMath	8196	0	4489680	0.18	0.00
A	StringSearch	1125	0	300710	0.37	0.00
L	CRC32	51	0	11296	0.48	0.00
	Blowfish	26184	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

the appropriate choice, where only the sequence of instructions performing encryption has *mult*, *div* instructions separated by five instructions.

$$\begin{aligned} W_{rt}(d1) &= 1/d1 \\ W_{rt}(d3) &= 1/d3 \\ W_{rt}(d2) &= \begin{cases} 0 & \text{if } d2 > c, \text{where c} \in \mathbb{R}, \\ 1/d2 & \text{if } d2 \leq c \end{cases} \end{aligned} \tag{10}$$

Figure 9(b) shows the threshold-concomitance analysis on RSA to find the appropriate distance between $mult - div$ segments using Equation (8) and Equation (10). The distance between two consecutive $mult - div$ segments (referred as $d2$ in Equation (10) and Figure 9(b)) is ignored ($W_{rt}(d2)$=0) if it is greater than the threshold distance c, which is increased from one upwards. As Figure 9(b) depicts, the concomitance stabilizes at a distance of 25, hence having a threshold-concomitance of 25.

As per the concomitance analysis, the XOR instructions (repeated within a window of 85 instructions) are mostly used in encryption routines (aka critical segments) of a cryptographic program. A multiplication then the division within a window size of five is another signature that stands out in RSA. Table I shows the number of *xor hits* and Multiplication and Division hits for different benchmarks. The first column of Table I divides the benchmarks into general and cryptographic programs. The second column details the name of applications, the third column gives the number of *xor*s in the application, and the fourth gives the number of *mult* and *div* instructions within



```
Encrypt:
 sw $2,8($30)
  lw $2,4($30)
  lw $3,8($30)
  xor $2,$2,$3
  sw $2,4($30)
  lw $3,8($30)
  sll $2,$3,0x4
  lw $3,0($30)
  xor $2,$3,$2
  sw $2,0($30)
  lw $3,0($30)
 srl $2,$3,0x10
…
…
…
    .end Encrypt
```

```
Encode:
lw $2,0($30)
lw $3,16($30)
mult $2,$3
mflo $2
sw $2,0($30)
lw $2,0($30)
lw $3,32572($28)
div $0,$2,$3
bne $3,$0,400
break
addiu $1,$0,-1
bne $3,$1,400938
lui $1,32768
bne $2,$1,400938
break
mfhi $2
```

Fig. 10. Signature definitions.

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* that occur in the trace. The total *xor* and the *m hits* were obtained by using the SimpleScalar™ instruction set simulator.

The percentage *xor hits* (% X) for general programs (noncryptographic programs) are much less than for the cryptographic programs as shown in Table I. All analyzed cryptographic programs, except RSA, have significant *xor hits* due to the usage of *XOR* instructions for encryption. As Table I 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 I. No other program except RSA has *m hits*. RSA has an *m hit* percentage of 0.12% on the total number of instructions executed.

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

4.1.1. autoRIJID: Signature 1—sigXOR. The first signature (*sigXOR*) is defined as shown in the diagram on the left side of Figure 10. When an *XOR* instruction is executed for the first time, it is identified as the start of the signature. The signature expires when there is no more *XOR* instructions seen before 85 instructions. The value of 85 is decided based on our concomitance analysis and the findings from the research presented in Gordon-Ross and Vahid [2003].

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*.

4.1.2. autoRIJID: Signature 2—sigMULTDIV. RSA algorithm is an exception that does not use XOR within its encoding instruction sequence. Instead, it uses *MULT* and *DIV* instructions. Therefore a signature is defined as shown in the right side diagram of Figure 10, 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

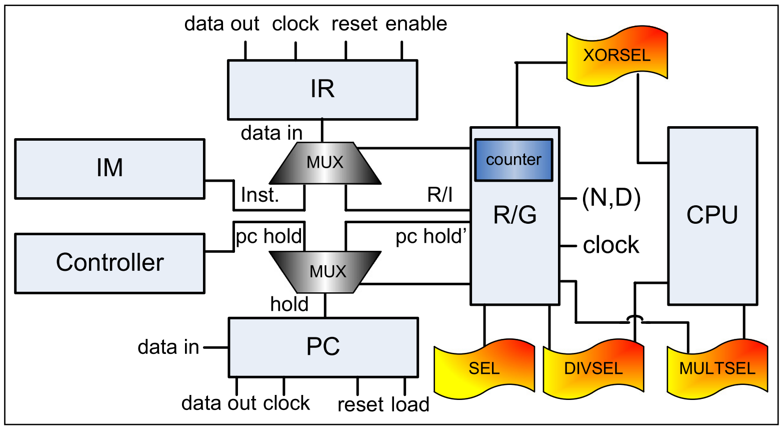


Fig. 11. *autoRIJID* architecture.

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

4.2. Hardware Architecture

The *autoRIJID* architecture is shown in Figure 11, which includes the random generator (R/G) that does 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. The R/G does both read and write to $XORSEL$.

Two similar flags ($DIVSEL$ and $MULTSEL$) are used for *MULT* (multiplication) and *DIV* (division) instructions to identify the *sigMULTDIV* signature that was explained in Section 4.1.2. The R/G sets and resets *SEL* flag based on the values inside $DIVSEL$ and $MULTSEL$ that are set by CPU, when the corresponding instructions execute.

The random instruction injection is performed in *autoRIJID* as the same way as explained for *softRIJID* in Section 3.2, by holding *PC* and injecting instructions into *IR*. Registers for the injected instructions are chosen random, while preserving the state of the processor. The random instruction generation performed by R/G is limited by the *injection pair* as explained earlier in Section 3.2. When one signature is detected (*sigXOR/sigMULTDIV*), the system does not detect the other. Hence this implementation avoids nested combinations.

We acknowledge that the signature detection has limited usefulness, since it might be easier for an expert to tag the code as we suggest for *softRIJID*. However, there are customers who attempt to use off-the-shelf cryptographic programs that are not written by security experts. Our signature detection unit at least guarantees that the cryptographic program is protected against side channel attacks without no modification to the code if the customer can afford an extra minuscule of hardware overhead to attach the signature detection unit.

5. DESIGN FLOW

This section explains the software design flow of *softRIJID* and the hardware design flows for both *softRIJID* and *autoRIJID*.

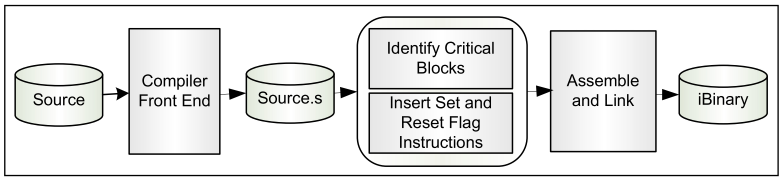
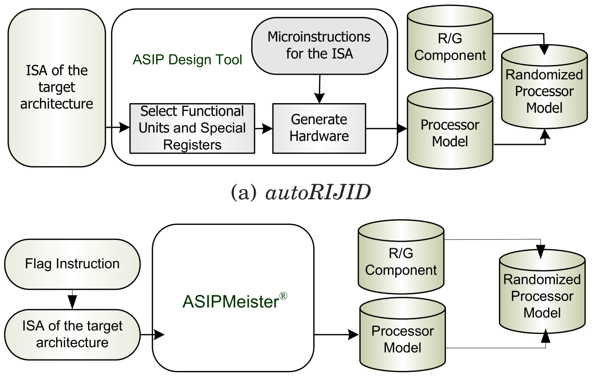


Fig. 12. Software design flow.


ISA of the target architecture
ASIP Design Tool
Microinstructions for the ISA
Select Functional Units and Special Registers
Generate Hardware
R/G Component
Processor Model
Randomized Processor Model
Flag Instruction
ISA of the target architecture
ASIPMeister®
R/G Component
Processor Model
Randomized Processor Model

(a) *autoRIJID*

(b) *softRIJID*

Fig. 13. Hardware design flow.

5.1. Software Design Flow of softRIJID

Figure 12 depicts the software design flow of the *softRIJID* 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 routines in the assembly code are tagged as explained in Section 3.1. The resulting assembly file is assembled and linked to generate the binary of the application (*iBinary* in Figure 12). Even though the instrumentation process here is performed manually, it is possible to automate it as explained in Section 3.1.

5.2. Hardware Design Flow

Figure 13 depicts the generation of processor models that implement *softRIJID* and *autoRIJID* frameworks in a pipelined RISC processor.

As shown in Figure 13(b), an additional flag instruction, which is used as a tag to enable and disable randomization in *softRIJID*, 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* [PEAS Team 2002]) in Figure 13(b) to generate the *softRIJID* processor model.

The hardware design flow for *autoRIJID* as shown in Figure 13(a) is very similar to the flow of *softRIJID*, except that no special instruction (or Flag) is required. Necessary functional units and special registers for signature detection are selected for *autoRIJID*. The microinstructions and the functional units are combined to generate a hardware processor model.

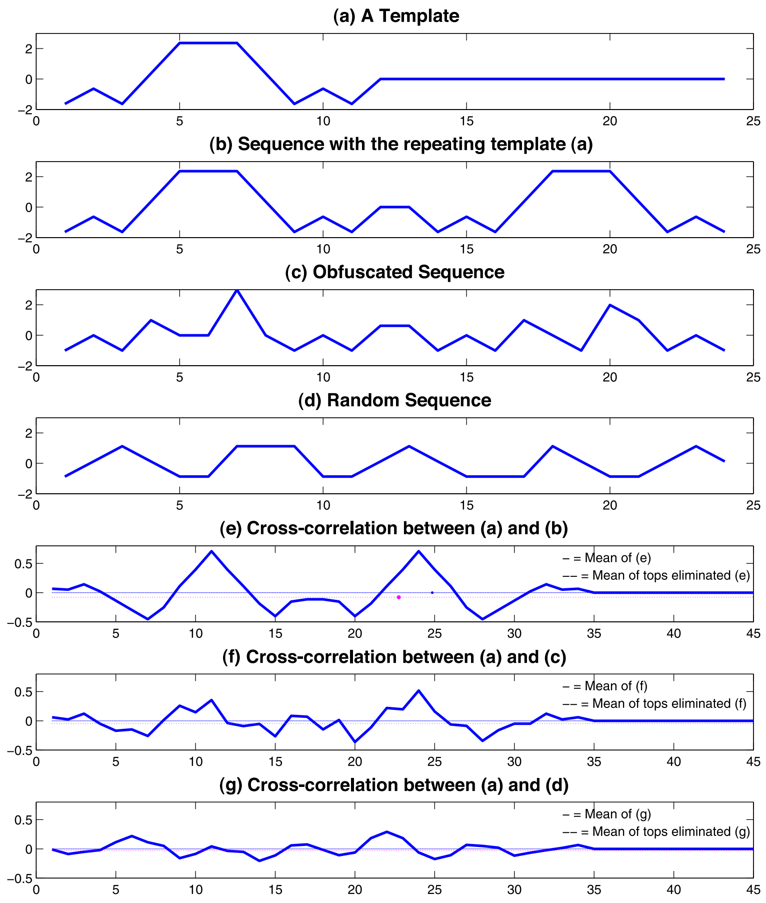


Fig. 14. An example *RIJIDindex* calculation.

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 (implemented using Linear Feedback Shift Register (LFSR)) and then is combined into both *softRIJID* and *autoRIJID* processors. For our experiments, we generated unique seeds for R/G using the *date* command in Linux. However, in real implementations, several environmental properties like temperature, clock or noise can be sensed to find a unique seed for the random generator at each run.

6. RIJIDINDEX

In our framework, *RIJIDindex* is used as the measure to evaluate the "*randomness*" provided by *RIJID* to scramble a power profile. *RIJIDindex* is a metric for predicting the vulnerability of a power profile. Therefore, it could be utilized instead of performing power analysis attacks to measure the vulnerability. In *RIJID* framework we: (1) analyze the original power sequence and extract a template; (2) apply *RIJID* to the application and measure the scrambled power sequence; and (3) use *RIJIDindex* as a measure to compute the "randomness" provided by *RIJID* in the scrambled power sequence. The rest of this section discusses how we came up with *RIJIDindex*.

Figure 14 depicts an example of how *RIJIDindex* is computed using cross-correlation [Bourke 1996]. When a single occurrence (the template as shown in

Figure 14(a)) of a repeating sequence is cross-correlated with the original sequence (as shown in Figure 14(b)), significant peaks will appear in the output at places where the template matches with the original sequence as depicted in Figure 14(e). Figure 14(c) depicts the obfuscated sequence, that is the scrambled sequence using *RIJID*. Figure 14(d) depicts a random sequence that does not have the template. As depicted in Figures 14(f) and 14(g), the cross-correlation between the template and either the obfuscated sequence or the random sequence do not produce significant peaks.

Here we perform a process we call *top elimination*, where we remove the significant peaks (decided based upon the original power profile, two in our example in Figure 14) from all three correlated waveforms. The mean values of all three correlated waveforms, before and after the *top elimination* are shown in Figures 14(e)–(g) with continuous and dotted lines respectively. As expected, the distance between the continuous and dotted lines is the highest in Figure 14(e) (correlated with the original sequence), the lowest in Figure 14(g) (correlated with the random sequence) and in between in Figure 14(e) (correlated with the obfuscated sequence).

Equation (11) defines 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 that cross-correlate. Equation (12) gives the mean value of the resulting cross-correlated sequence with a number of peaks or maximum values removed. $TopT$ represents T number of maximum values in sequence $(f * g)_i$. Equation (13) defines the mean distance of the sequence f. The number of significant peaks are decided based on the power profile of the critical instruction sequence or number of occurrences of the template to be specific. For example, TripleDES has 16 rounds within the sequence of instructions performing encryption. Therefore, when the template is correlated with a power sequence of instructions performing encryption, there will be 16 significant peaks in the cross-correlated wave. Thus the number of significant peaks is 16.

$$\Psi_{f,g} = \frac{\sum_{i=1}^{2N-1}(f * g)_i}{(2N - 1)} \tag{11}$$

$$\varphi_{f,g,T} = \frac{\sum_{i=1}^{2N-1}(f * g)_i - \sum_1^T TopT}{(2N - 1) - T} \tag{12}$$

$$\Delta_f = \Psi_{f,g} - \varphi_{f,g,T} \tag{13}$$

$$RIJIDindex = (\Delta_o - \Delta_z)/(\Delta_o - \Delta_r). \tag{14}$$

Now, we use these distances between means to define *RIJIDindex*. *RIJIDindex* will give us a measure of how much the vulnerable (original) sequence with a template is obfuscated compared to a random sequence. When the distance between means of obfuscated sequence reaches the distance between means of the random sequence, the vulnerability of the obfuscated sequence reduces (that is, the scrambling imposed on the original sequence increases). Hence, *RIJIDindex* is defined as in Equation (14), using the distances between means of the original, obfuscated and random sequences based on a specific template from the original sequence. *RIJIDindex* ($0 \leq RIJIDindex \leq 1$), as defined in Equation (14), uses the distance between means of the original (Δ_o), obfuscated (Δ_z) and random (Δ_r) sequences. *RIJIDindex* reaches one when the distance between means of the obfuscated sequence equals the distance between means of the random sequence. Such case gets the best scrambling from *RIJID* processor, where the dissipated power sequence appears as a random sequence (that is, no expected templates exist). The higher the *RIJIDindex* of a power sequence, the higher the masking.

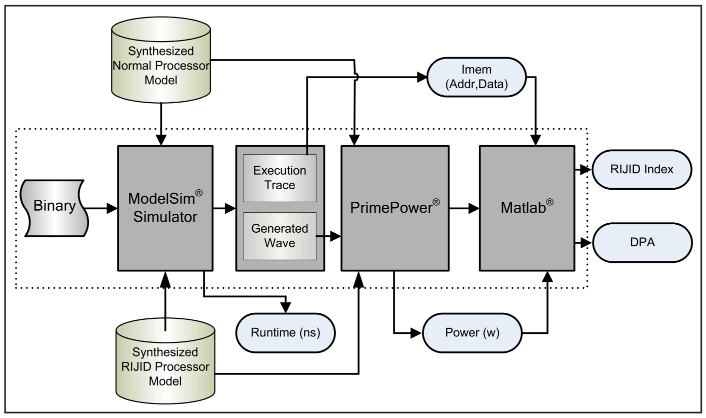


Fig. 15. Measurement setup.

7. EXPERIMENTAL SETUP

In this section, the main components used for experiments and the randomness testing are explained. Both *softRIJID* and *autoRIJID* frameworks are implemented with PISA instruction set (as implemented in SimpleScalar™ tool set with a six stage pipeline) processor without cache. Figure 15 depicts the process of measuring *RIJIDindex* for an application trace. Applications in C are compiled using GNU/GCC® cross-compiler for the PISA instruction set. ASIPMeister[PEAS Team 2002], an automatic ASIP design tool is used to generate synthesizable VHDL model of the processor as explained in Section 5.2. The processor models are synthesized using Synopsys Design Compiler.

In *softRIJID*, original binary is produced from the original code of the program and simulated with the normal processor model, which does not have *RIJID* and the instrumented binary is produced as explained in Section 5.1 and simulated with the *softRIJID*processor model. It is worth noting the *autoRIJID*model does not require software instrumentation. Corresponding binaries and synthesized processor models are simulated together in ModelSim HDL simulator, which generates the stimulus wave with switching information. Using ModelSim simulator, the execution trace is verified for correctness 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 15. 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. *RIJIDindex* from the power sequences is calculated by implementing necessary functions in Matlab. Power Analysis (DPA) for *autoRIJID* is implemented in a C program, where specific instruction power values are extracted and used.

The experiments demonstrated in this article are performed for the applications implemented in C, which are taken from MiBench embedded benchmark suite [Guthaus et al. 2001]. Dissipated power waves are observed by running the encryption programs for specific keys and different data.

8. RESULTS

This section presents the experimental results of both *softRIJID* and *autoRIJID* processor models. *RIJIDindex* values are displayed for both processors and a DPA is presented for *autoRIJID*.

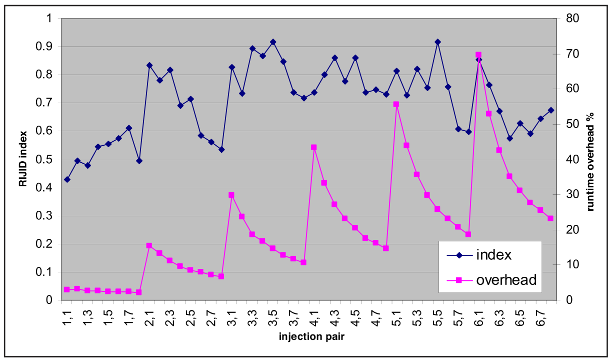


Fig. 16. *RIJIDindex* and runtime overhead on TripleDES.

8.1. *softRIJID*

8.1.1. RIJIDindex Vs. Runtime Overhead. Figure 16 shows the variation of *RIJIDindex* 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.

In Figure 16, the maximum *RIJIDindex* values of 0.9158 and 0.9157 are produced for the injection pair (3,5) and (5,5) with runtime overheads of 14.5% and 25.9% respectively. When the (N,D) pair is small, the possible combinations of injections are limited 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 compared to (N,D)= (3,5).

$$C = \left(\sum_{k=1}^{N} \frac{6!}{(6-k)!} \right)^{\frac{115}{D}}. \tag{15}$$

Equation (15) defines 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 that 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. 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.

8.1.2. softRIJID on Other Cryptographic Applications. *softRIJID* is applied to different encryption programs and the results are tabulated in Table II, where column 1 and 2 gives the N, D pair; columns 3, 5, 7 give the *RIJIDindex* for RSA, IDEA, and RC4 algorithms; and columns 4, 6, 8 and refer the runtime overheads. The results show that RSA gets the highest *RIJIDindex* 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 II.

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

Table II. Runtime Overheads and $RIJIDindex$ for Cryptographic Programs

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 III. Energy Overhead for Injection Pair (5,5)

	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 IV. Hardware Overheads

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

8.1.3. Energy Overhead. The average energy values for the original and scrambled (with *injection pair* (5,5)) encryption routines for different encryption programs are tabulated in Table III. Since the maximum scrambling takes place using an *injection pair* value of (5,5) as presented in Table II, the average energy values are calculated for (5,5) and the overheads are computed against the values of the original encryption instruction sequence.

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

8.1.4. Hardware Overhead. An additional functional unit is created for the random instruction generation (R/G as explained in Section 3.2). Our *softRIJID* processor model costs an additional 1.98% area overhead when compared with the normal processor model, as shown in Table IV. The clock period remains the same for *softRIJID* compared to the original processor.

softRIJID consumes far less additional hardware when compared to other hardware designs, such as the secure coprocessor proposed in Tiri et al. [2005] that costs three times increase in area.

8.1.5. Varying the Instruction Injection. Figure 17 depicts the variation of power values when using the same random instruction for injection, but using: a) immediate operands that are set to zero; b) registers that do not change during the injection for a given type of random instruction (thus, injected ADD will always work with a fixed register); and c) random registers and random immediate operand in the injected instructions.

When similar instructions are injected next to each other (circled in Figure 17) due to the values scheduled by the random generator, a constant power value is tend to be seen for the zero register and fixed register cases. This is because of insufficient

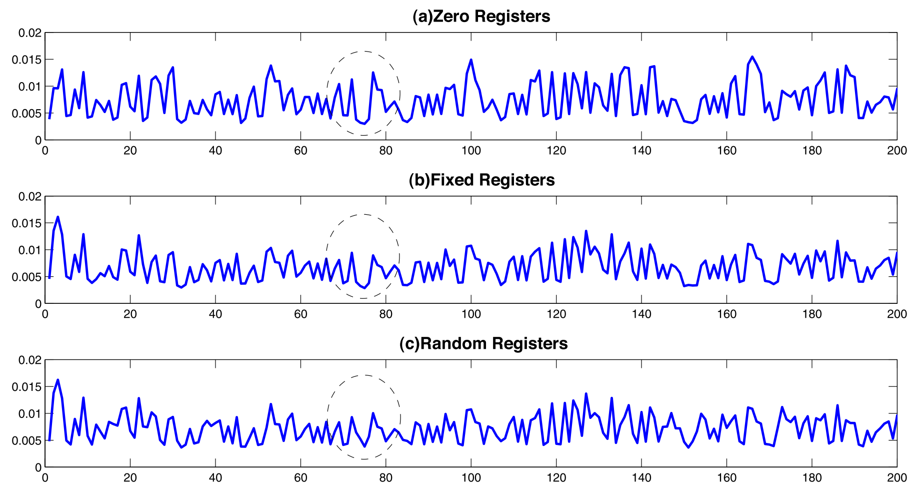


Fig. 17. Register types on injected instructions.

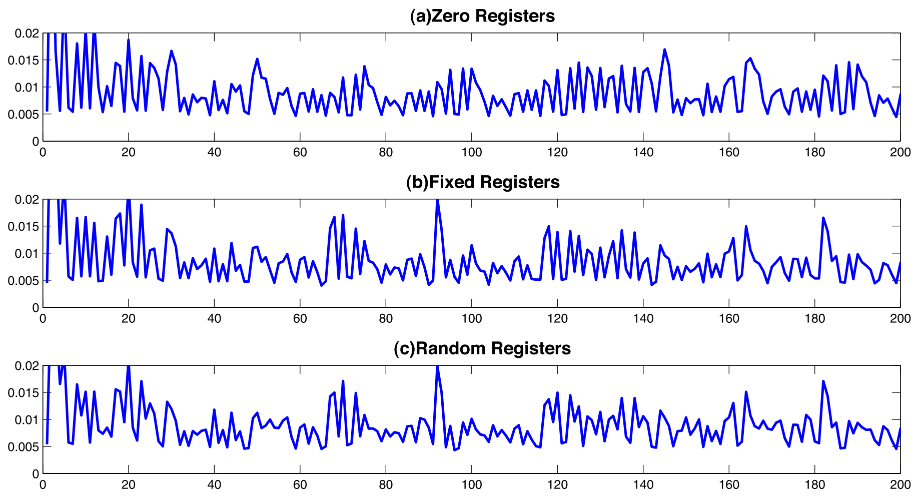


Fig. 18. Register types on injected instructions for different seed.

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 18 shows the power profile for the same instruction segment as the segment shown in Figure 17 with a different seed value. As visible from the figures, the power profile significantly varies when changing 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.

8.2. *autoRIJID*

For *autoRIJID* technique, evaluations are performed on programs using three different processors: (1)*sigXOR* **processor**, identifying *sigXOR* signature; (2) *sigMULTDIV*

Table V. Runtime Overheads for *autoRIJID* Processors

		Normal (μs)	**sigXOR** (μs)	**% O.H**	**sigMULTDIV** (μs)	**% O.H**	***autoRIJID*** (μs)	**% O.H**	**% E. 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

processor, identifying *sigMULTDIV* signature; and (3) ***autoRIJID*** **processor**, identifying both *sigXOR* and *sigMULTDIV* signatures.

8.2.1. Runtime and Energy. Table V depicts the runtime and energy overheads of signature processors for different benchmarks. The first column of Table V 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 *autoRIJID* 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 *autoRIJID* is applied.

The runtime overheads when using *sigXOR* is much larger for cryptographic programs compared to noncryptographic programs as shown in Table V. 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 I. This XOR is outside the encryption instruction sequence, yet due to RSA's small code size, we get a large overhead. Table V shows that *sigXOR* does not significantly affect noncryptographic programs (except CRC32) if we apply *autoRIJID*. The maximum runtime overhead for noncryptographic programs is for StringSearch with just 1.9%. The *autoRIJID* consumes 13.1% of runtime when applied to RSA as shown in Table V. 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 V, 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%.

Table VI. Hardware Summary

Processor	**Area** (*cell*)	**Clock** (*ns*)	**Area Overhead** (%)
Normal	111,188	41.33	N/A
sigXOR	112,123	41.69	0.8
sigMULTDIV	112,200	41.69	0.9
autoRIJID	112,545	41.69	1.2

Table VII. *RIJIDindex* Using *autoRIJID*

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

8.2.2. Hardware Summary. Table VI depicts the hardware overheads of signature processors. The first column of Table VI denotes the types of processors used. The second column states the area of each processor. The clock period for each processor model is listed in the third column. The area overheads are presented in the fourth column. The processor area is smaller for *sigXOR*, *sigMULTDIV* and *autoRIJID* compared to *softRIJID*, which is shown in Table IV because of no special instruction usage.

The clock period does not have any significant difference when signature recognition is implemented. *autoRIJID* 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*.

8.2.3. Obfuscation and the RIJIDindex. The higher the *RIJIDindex* the higher the scrambling provided and lower the vulnerability of the power sequence. Table VII depicts the *RIJID*indices of cryptographic programs and their loop size in number of instructions when *autoRIJID* is imposed. The *injection pair* of (5,5) is imposed on *autoRIJID*, where (5,5) is considered the appropriate choice for our implementation as explained in Figure 16. 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.

8.2.4. Power Analysis Attack and autoRIJID. Figure 19 depicts the DPA performed on TripleDES (based on the technique explained in Brier et al. [2004]), an attempt to prove that our method prevents DPA. TripleDES was chosen because it provides the lowest *RIJIDindex*, thus the most vulnerable amongst the programs shown above. Plots are provided for each selection bit where the last *sbox* lookup of the 16*th* round is chosen as the attacking point. Figure 19(a) shows DPA plots on TripleDES without the *autoRIJID* implementation and Figure 19(b) shows using *autoRIJID* applied.

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. The TripleDES implementation

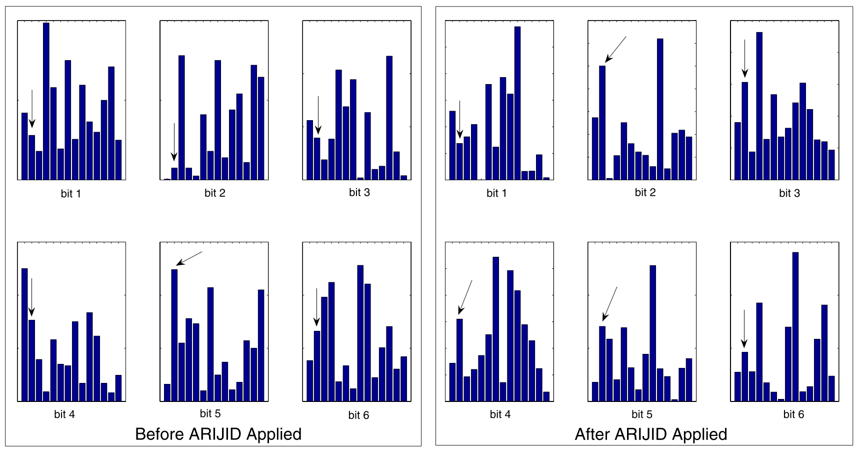


Fig. 19. DPA before and after *autoRIJID*.

we considered has three level DES encryption with 56 bit keys each. Each key has 8 parity bits forming a 64bit block used for computation.

As the left half of Figure 19 depicts, the key is successfully predicted (place of the correct key is pointed to by arrows in each plot) when using *bit 5* (the second plot in the second row) of the 6 selection bits used for SBOX lookup. All the other bits except *bit 5* (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 19.

The key cannot be predicted using any of the bits after *autoRIJID* is applied as shown in the right half of Figure 19, where no peaks appear on the correct key.

9. CONCLUSIONS

In this article, we have presented a randomized instruction injection technique (*RIJID*) to prevent power analysis attack. A random number of real instructions are injected at random places during the runtime of the processor to scramble the power profile, so that adversaries cannot extract any useful information. Two different *RIJID* processors (*softRIJID* and *autoRIJID*) are implemented and evaluated. While *softRIJID* uses special instructions to tag critical routines of cryptographic applications, *autoRIJID* utilizes predefined signatures to automatically identify them. The *softRIJID* processor is useful when a lesser number of hardware modifications is preferred and tagging is considered easier. *autoRIJID* reduces the onus on the programmer by taking care of obfuscation of the power profile, but with minor hardware modifications. Therefore, *autoRIJID* could be applied to native binaries as well.

A new metric (*RIJIDindex*) is introduced to measure the degree of randomization and therefore the suitable injection values can be identified. *softRIJID* implementation consumes an area overhead of 1.98%, an average runtime overhead of 29.8% and an average energy overhead of 27.1%, while *autoRIJID* consumes 1.2%, 25.0%, and 28.5% in area, runtime, and energy overheads respectively.

Although *RIJID* is proposed as a countermeasure for power analysis attacks, it can be used to prevent other side channel attacks such as electro magnetic analysis (SEMA and DEMA). Future work includes designing a methodology to work with super-scalar and VLIW processor architectures. FPGA prototyping will be performed to justify the practicality of our approach.

REFERENCES

AKKAR, M.-L., BEVAN, R., DISCHAMP, P., AND MOYART, D. 2000. Power analysis, what is now possible... In *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT'00)*. Springer, 489–502.

AMBROSE, J. A., RAGEL, R. G., AND PARAMESWARAN, S. 2007a. RIJID: Random code injection to mask power analysis based side channel attacks. In *Proceedings of the IEEE/ ACM Design Automation Conference*. ACM Press, New York, NY, 489–492.

AMBROSE, J. A., RAGEL, R. G., AND PARAMESWARAN, S. 2007b. A smart random code injection to mask power analysis based side channel attacks. In *Proceedings of the the International Conference on Hardware/Software Codesign and System Synthesis*. ACM Press, New York, NY, 51–56.

BARBOSA, M. AND PAGE, D. 2005. On the automatic construction of indistinguishable operations. In *Proceedings of the 10th IMA International Conference on Cryptography and Coding,* Lecture Notes in Computer Science. vol. 3796, 233–247.

BLAKE, I., SEROUSSI, G., SMART, N., AND CASSELS, J. W. S. 2005. *Advances in Elliptic Curve Cryptography* (London Mathematical Society Lecture Note Series). Cambridge University Press.

BOUBEKEUR, T. AND SCHLICK, C. 2007. *GPU Gems 3 - Generic Adaptive Mesh Refinement*. Addison-Wesley, Boston, MA, Chapter 5, 93–104.

BOURKE, P. 1996. Cross Correlation: AutoCorrelation – 2D Pattern Identification. http://astronomy.swin.edu.au/pbourke/other/correlate/index.html.

BRIER, E., CLAVIER, C., AND OLIVIER, F. 2004. Correlation power analysis with a leakage model. In *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems,* Lecture Notes in Computer Science, vol. 3156. 16–29.

BRUMLEY, D. AND BONEH, D. 2003. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, Berkeley, CA.

CHARI, S., JUTLA, C. S., RAO, J. R., AND ROHATGI, P. 1999. Towards sound approaches to counteract power-analysis attacks. In *Proceedings of the International Cryptology Conference*. Springer, 398–412.

CLAVIER, C., CORON, J.-S., AND DABBOUS, N. 2000. Differential power analysis in the presence of hardware countermeasures. In *Proceedings of the 2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES'00)*. Springer, 252–263.

CORON, J. 1999. Resistance against differential power analysis for elliptic curve cryptosystems. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 292–302.

CORON, J.-S. AND GOUBIN, L. 2000. On Boolean and arithmetic masking against differential power analysis. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 231–237.

DAEMEN, J. AND RIJMEN, V. 1999. Resistance against implementation attacks: A comparative study of the AES proposals. http://csrc.nist.gov/CryptoToolkit/aes/round1/pubcmnts.htm.

GEBOTYS, C. 2006. A table masking countermeasure for low-energy secure embedded systems. *IEEE Trans. VLSI Syst. 14,* 7, 740–753.

GEBOTYS, C. H. AND GEBOTYS, R. J. 2003. Secure elliptic curve implementations: An analysis of resistance to power-attacks in a DSP processor. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*. Springer, 114–128.

GORDON-ROSS, A. AND VAHID, F. 2003. Frequent loop detection using efficient non-intrusive on-chip hardware. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'03)*. ACM Press, New York, NY, 117–124.

GOUBIN, L. AND PATARIN, J. 1999. DES and differential power analysis (The "Duplication" Method). In *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*. Springer, 158–172.

GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC'01).* IEEE Computer Society, Washington, DC, 3–14.

HWANG, D. D., SCHAUMONT, P., TIRI, K., AND VERBAUWHEDE, I. 2006. Securing embedded systems. *IEEE Secur. Privacy 4,* 2, 40–49.

IRWIN, J., PAGE, D., AND SMART, N. P. 2002. Instruction stream mutation for non-deterministic processors. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02)*. IEEE Computer Society, Washington, DC, 286.

JANAPSATYA, A., IGNJATOVIC, A., AND PARAMESWARAN, S. 2006. Exploiting statistical information for implementation of instruction scratch memory in embedded system. *IEEE Trans. VLSI Syst. 14,* 8, 816–829.

JOYE, M., PAILLIER, P., AND SCHOENMAKERS, B. 2005. On Second-Order Differential Power Analysis. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 293–308.

KOCHER, P., JAFFE, J., AND JUN, B. 1999. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference,* Lecture Notes in Computer Science, vol. 1666, 388–397.

MACÉ, F., STANDAERT, F.-X., AND QUISQUATER, J.-J. 2007. Information theoretic evaluation of side-channel resistant logic styles. In *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'07)*. Springer, 427–442.

MANGARD, S. 2003. A simple power-analysis (spa) attack on implementations of the aes key expansion. In *Proceedings of the International Conference on Information Security and Cryptology*. P. J. Lee and C. H. Lim Eds. Lecture Notes in Computer Science Series, vol. 2587. Springer, 343–358.

MAY, D., MULLER, H. L., AND SMART, N. P. 2001a. Non-deterministic Processors. In *Proceedings of the 6th Australasian Conference on Information Security and Privacy (ACISP'01)*. Springer, 115–129.

MAY, D., MULLER, H. L., AND SMART, N. P. 2001b. Random register renaming to foil DPA. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 28–38.

MAYER-SOMMER, R. 2000. Smartly Analyzing the simplicity and the power of simple power analysis on smartcards. In *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 78–92.

MERTEN, M. C., TRICK, A. R., BARNES, R. D., NYSTROM, E. M., GEORGE, C. N., GYLLENHAAL, J. C., AND HWU, W. 2001.An architectural framework for runtime optimization. *IEEE Trans. Comput. 50,* 6, 567–589.

MESSERGES, T. S., DABBISH, E. A., AND SLOAN, R. H. 1999. Power analysis attacks of modular exponentiation in smartcards. In *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES'99)*. Springer, 144–157.

MESSERGES, T. S., DABBISH, E. A., AND SLOAN, R. H. 2002. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Comput. 51,* 5, 541–552.

MURESAN, R. AND GEBOTYS, C. H. 2004. Current flattening in software and hardware for security applications. In *Proceedings of the the International Conference on Hardware/Software Codesign and System Synthesis*. ACM Press, New York, 218–223.

NOVAK, R. 2002. SPA-based adaptive chosen-ciphertext attack on RSA implementation. In *Proceedings of the 5th International Workshop on Practice and Theory in Public Key Cryptosystems (PKC'02)*. Springer, 252–262.

ORS, S. B., GURKAYNAK, F., OSWALD, E., AND PRENEEL, B. 2004. Power-analysis attack on an ASIC AES implementation. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)*. 546.

OSWALD, E., MANGARD, S., HERBST, C., AND TILLICH, S. 2006. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In *Proceedings of the Cryptographers' Track at the RSA Conference*. D. Pointcheval Ed. Lecture Notes in Computer Science Series, vol. 3860. Springer, 192–207.

PEAS TEAM. 2002. ASIP Meister. Available at http://www.eda-meister.org/asip-meister/.

QUISQUATER, J. AND SAMYDE, D. 2001. ElectroMagnetic Analysis (EMA): Measures and counter-measures for smart cards. In *E-smart*. Springer, 200–210.

RATANPAL, G. B., WILLIAMS, R. D., AND BLALOCK, T. N. 2004. An on-chip signal suppression countermeasure to power analysis attacks. *IEEE Trans. Depend. Secure Comput. 01,* 3, 179–189.

RAVI, S., RAGHUNATHAN, A., AND CHAKRADHAR, S. 2004. Tamper resistance mechanisms for secure, embedded systems. In *Proceedings of the International Conference on VLSI Design*. 605.

REGAZZONI, F., CEVRERO, A., STANDAERT, F.-X., BADEL, S., KLUTER, T., BRISK, P., LEBLEBICI, Y., AND IENNE, P. 2009. A design flow and evaluation framework for dpa-resistant instruction set extensions. In *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'09)*. Springer-Verlag, Berlin, Heidelberg, 205–219.

SAPUTRA, H., VIJAYKRISHNAN, N., KANDEMIR, M., IRWIN, M. J., BROOKS, R., KIM, S., AND ZHANG, W. 2003. Masking the energy behavior of DES encryption. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe*. 10084.

STANDAERT, F.-X., PEETERS, E., AND QUISQUATER, J.-J. 2005. On the masking countermeasure and higher-order power analysis attacks. In *Proceedings of the International Conference on Information Technology: Coding and Computing*. IEEE, 562–567.

TIRI, K., HWANG, D., HODJAT, A., LAI, B., YANG, S., SCHAUMONT, P., AND VERBAUWHEDE, I. 2005. A side-channel leakage free coprocessor IC in 0.18μm cmos for embedded AES-based cryptographic and biometric processing. In *Proceedings of the IEEE/ACM Design Automation Conference*. ACM Press, New York, NY, 222–227.

TRICHINA, E., SETA, D. D., AND GERMANI, L. 2003. Simplified Adaptive Multiplicative Masking for AES. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*. Springer, 187–197.

WADDLE, J. AND WAGNER, D. 2004. Towards efficient second-order power analysis. In *Proceedings of the 1st International Workshop on Cryptographic Hardware and Embedded Systems,* Lecture Notes in Computer Science, vol. 3156. 1–15.

Received December 2009; revised June 2010, November 2010; accepted July 2011