

Mitigating Power Attacks through Fine-Grained Instruction Reordering

Yun Chen*

National University of
Singapore

Andreas Diavastos
Universitat Politècnica de
Catalunya

Ali Hajiabadi*

National University of
Singapore

Shivam Bhasin
Temasek Laboratories,
Nanyang Technological
University

Romain Poussier

Nanyang Technological
University

Trevor E. Carlson
National University of
Singapore

ABSTRACT

Side-channel attacks are a security exploit that take advantage of information leakage. They use measurement and analysis of physical parameters to reverse engineer and extract secrets from a system. Power analysis attacks in particular, collect a set of power traces from a computing device and use statistical techniques to correlate this information with the attacked application data and source code. Countermeasures like just-in-time compilation, random code injection and instruction descheduling obfuscate the execution of instructions to reduce the security risk. Unfortunately, due to the randomness and excess instructions executed by these solutions, they introduce large overheads in performance, power and area.

In this work we propose a scheduling algorithm that dynamically reorders instructions in an out-of-order processor to provide obfuscated execution and mitigate power analysis attacks with little-to-no effect on the performance, power or area of the processor. We exploit the time between operand availability of critical instructions (*slack*) to create high-performance random schedules without requiring additional instructions or static prescheduling. Further, we perform an extended security analysis using different attacks. We highlight the dangers of using incorrect adversarial assumptions, which can often lead to a false sense of security. In that regard, our advanced security metric demonstrates improvements of $34\times$, while our basic security evaluation shows results up to $261\times$. Moreover, our system achieves performance within 96% on average, of the baseline unprotected processor.

1. INTRODUCTION

Timing, electromagnetic (EM) and power analysis attacks, also called side-channel attacks, exploit physical parameters of the processor to extract secret information from a running program in the system. Power analysis attacks are the most common as they are relatively easy and cheap to execute as they do not require any special equipment or knowledge of the internal design of the system. Power analysis attacks take advantage of the synchronization and high correlation

of the power consumption with the instructions and data being processed to identify patterns in the victim’s program behavior. These patterns are then analyzed to reverse engineer the program execution and extract secret information [18].

Many countermeasures have been proposed to combat power analysis attacks in all levels of the design stack [2, 3, 14, 17, 23, 25, 29, 40, 50]. We categorize these countermeasures into two generic mitigation techniques: (1) *hiding* and (2) *masking*. *Hiding* countermeasures limit or hide the information available to an adversary. In other words, *hiding* lowers the available signal-to-noise ratio (SNR) to an adversary. SNR suppression can be achieved by balancing or suppression techniques where relations between power consumed are weakened for executed data or instruction (e.g. power balancing, physical signal suppression, noise amplification) or by randomizing the execution sequence (e.g. jitter, randomization of execution order). Thus, *hiding* is a system-wide solution that alters the power consumption of the system in a way that hides the actual power consumed by the instructions and the data being processed.

Masking, on the other hand, splits any sensitive intermediate variable into several statistically independent shares, similar to the principles of Shamir’s secret sharing [43]. An adversary can learn nothing about the sensitive intermediate variable, unless all the shares are available. As the shares are processed independently, *masking* effectively removes all leakage dependencies for the lowest statistical moments, which increases the attack complexity. Often, design constraints must be put in place to guarantee independence of shares in data processing. The processing of independent shares requires additional computation, leading to a non-negligible overheads. While the *masking* technique can provide strong guarantees from a cryptographic point of view, it is effective only in the presence of noise. Thus *masking* and *hiding* are complementary countermeasures, where *hiding* provides the ideal low SNR environment for *masking* to be effective.

In this work, we focus on solutions and methodologies to improve secret *hiding*. In fact, the current solutions that implement the *hiding* technique make significant sacrifices in a number of important areas. Some significantly increase

*Yun Chen and Ali Hajiabadi contributed equally to this work.

the performance, power or area of the system to mitigate an attack [1, 3, 7, 17, 20, 24, 25]. The overheads in performance and power reach up to 300%, while the area can be twice the size of the original baseline core. Other solutions either lack generality [50] or require compiler support [10]. In general, we observe that no one single solution offers a competitive package that provides high-levels of security with minimal impact on the performance, power, design cost or compatibility of the system.

With this work, we aim to overcome these limitations of previous hiding techniques. We introduce a new countermeasure that leverages the scheduling information of dynamic instructions in an out-of-order processor to break the correlation to physical parameters with minimal overhead (3.7% on performance, 1.1% on power and 0.7% on area). To implement our proposed hiding technique in an efficient way, we propose an instruction scheduling technique that monitors the execution of an application and intelligently randomizes the issue of instructions to produce different instruction schedules in every loop without affecting the performance. More specifically, we add a low-cost structure (*Slack Unit*) in the processor pipeline that records the time difference between an instruction’s operand production (slack) and uses it to inject a random artificial delay (that is no larger than this original slack) to randomize the issue of non-critical instructions. The result is a low-overhead (both in energy and area, as well as performance), general-purpose and hardware-only technique that obfuscates instruction execution to break the power consumption correlation that is normally observed by the adversary.

In this paper we make the following contributions:

- A general-purpose out-of-order core design that provides a significant security improvement ($34\times$ with an advanced adversary, and $261\times$ when the attacker is using basic methods) against power analysis attacks to all executing applications with minimal power and area overheads (1.1% and 0.7% respectively);
- A dynamic instruction scheduling technique that intelligently combines instruction slack with randomness to avoid significant performance degradation (less than 3.7%), while strengthening the security of the system;
- An advanced, comprehensive security evaluation of the proposed solution. We present three different security evaluation techniques in an attempt to raise the bar for architectural security evaluation.

This paper is structured with an overview in Section 2 and describe the proposed architecture in Section 3. In Section 4 we outline our proposed evaluation methodology and in Section 6 we present our findings. In Section 7 we outline the state-of-the-art and conclude with Section 8.

2. MOTIVATION AND OVERVIEW

Power analysis attacks exploit the synchronization of instruction execution and its correlation with power consumption to uncover secret information from a running application. A power analysis attack is performed by collecting a set of power traces from a processor during victim application execution. The number of power traces required to reveal this

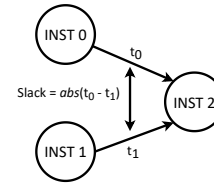


Figure 1: Computing slack for instructions. The slack for INST 2 is the absolute difference between the time that INST 0 and INST 1 produce their result.

secret information depends on how secure the processor is, where regular patterns can more easily be used to extract this data. Using statistical analysis, the adversary correlates the power consumption with the executed instructions and the data being processed to uncover the secret information of the application. Such attacks are effective due to the deterministic behavior of the processing units, and are possible because the adversary can reason about the processor’s behavior with a relatively small set of power measurements. As we show later, and as demonstrated by previous works [33], even modern general-purpose out-of-order processors demonstrate a high level of regularity, making them relatively easy to attack.

A common countermeasure for these types of attacks is to execute instructions in an unpredictable way that makes it difficult for the adversary to correlate the observed power with the code being executed. Many solutions have been proposed in this direction [1, 3, 7, 10, 17, 20, 24, 25, 50], however, most of them are application-specific, target simple dedicated-to-encryption hardware, or do not apply generically to processor platforms. Such solutions also tend to result in large performance and/or power overheads or require significantly more hardware to implement. One of the biggest challenges in designing secure processors is finding efficient generic solutions that can protect all applications without affecting application performance.

In this work, we focus on the general case and target high performance out-of-order general-purpose processors. To this end, we identify three major requirements needed to successfully design both a secure and efficient general-purpose processor: (1) desynchronize the instruction execution (create non-deterministic behavior) to increase the noise in the collected power traces without affecting the performance, (2) design a generic solution that allows for secure execution of all running applications, and (3) implement an efficient design that demonstrates minimal power and area overheads.

The key insight of this paper is that to efficiently offer security in a general-purpose processor without affecting its performance we must *randomize the ordering of instructions that are off the critical path of the execution*. As explained earlier, increase in security can result from randomizing the instruction execution order. In this work, we introduce randomization by delaying instruction issue time. Doing so desynchronizes the execution, enables non-deterministic behavior, and therefore increases the noise as seen in the power measurements. Consequently, the added noise makes it harder for the adversary to find the highly distinguishable statistical dependencies that will uncover secret information. To maintain baseline performance however, reordering must be

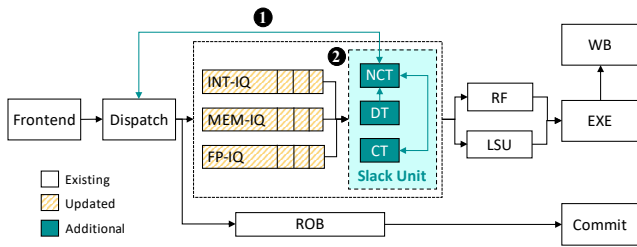


Figure 2: The microarchitecture of PARADISE.

done only on non-critical instructions. In this work, we define criticality by the time the input operands of an instruction are produced. In the example of Figure 1, instruction INST 2 has two input operands that are produced at different times. Assuming that INST 0 produces its output earlier than INST 1 then, in this pair of instructions, INST 0 is non-critical because delaying its execution until INST 1 produces its output will not change the starting time of INST 2. Delaying INST 1 however, will delay the execution of INST 2 therefore, we consider INST 1 a critical instruction.

Our proposed solution injects a delay only to non-critical instructions and within certain timing boundaries that will minimize performance overheads. We call this time margin *slack*. The slack is defined as the time difference between the generation of an instruction’s operands. In the example of Figure 1, INST 0 and INST 1 produce their data for INST 2 at time t_0 and t_1 respectively. The absolute difference of the two times ($abs(t_0 - t_1)$) is the slack for these two instructions. If we assume that $t_1 > t_0$, then INST 0 is considered to be non-critical for the execution of INST 2 and this information will be saved for the next time we execute this instruction. At that point, a random delay, not larger than the slack, will be introduced to increase the issue time of INST 0. This in turn increases instruction desynchronization, making it more difficult for an adversary to extract secret information.

Note that, although throughout this paper we refer to power analysis attacks only, the proposed technique is also applicable to other side channels that use similar attack methodologies like EM and timing attacks.

3. PARADISE MICROARCHITECTURE

In this section, we present an efficient general-purpose processor that detects the slack and randomly delays non-critical instructions to desynchronize the execution without affecting the critical path and successfully maintaining high performance. Non-critical instructions are delayed before being issued to the execution units to randomize their execution and their access to the register file or memory. To achieve this, we propose *PARADISE*, an out-of-order processor that implements a novel secure instruction scheduling policy.

Figure 2 shows the microarchitecture of PARADISE. It is built on top of SonicBoom [51], a RISC-V out-of-order core with a 7-stage pipeline. PARADISE implements a new structure called the Slack Unit that dynamically records the slack of instructions and communicates the appropriate delay to be injected to selected non-critical instructions. The Issue Queues (IQs) and the out-of-order scheduler are updated to communicate instruction information to the Slack Unit and

Destination Table (DT)		Critical Table (CT)	Non-Critical Table (NCT)		
PC (12 bits)	Non-critical PC offset (8 bits)	PC (12 bits)	PC (12 bits)	Slack (5 bits)	Stable (1 bit)
0x1310	192	0x1400	0x1210	10	0
0x1320	4	0x1410	0x1220	5	1
0x1330	8	0x1420	0x1230	0	1
...

Figure 3: The Non-Critical Table (NCT) stores non-critical instructions and their slack, the Critical Table (CT) stores critical instructions and the Destination Table (DT) stores the currently dispatched instructions.

delay the issuing of instructions that were previously injected with a delay respectively. These additions don’t affect the pipeline stages of the processor as they are parallel operations that execute seamlessly with the rest of the processor operations. Hence, PARADISE retains the frequency of the baseline SonicBoom core.

A dispatched instruction queries the Slack Unit, using its PC, and in case of a hit a delay, is injected to the instruction and the scheduler is notified that the current instruction will be issued with a delay (step ①). The delay is applied only after all its operands are produced. When an instruction is issued for execution, the Slack Unit stores its producers and the new slack between them (step ②).

3.1 Slack Unit

The Slack Unit is built using three set-associative memory structures to store different runtime information of the program, as shown in Figure 3. All structures use a Least Recently Used (LRU) replacement policy. The structures are:

- **Destination Table (DT):** Holds the instructions that are being issued and their non-critical producer. The DT is queried by issued instructions to check whether their non-critical producers are the same in different appearances;
- **Non-Critical Table (NCT):** Holds the non-critical instructions and their corresponding slack. For performance reasons, we also use these entries to indicate whether the instruction has been consistently a non-critical instruction (stable). If so, we use the stored slack as an upper bound for selecting a random delay for the corresponding instruction;
- **Critical Table (CT):** Holds a list of instructions that were marked as critical. This is necessary in order to handle criticality conflicts in cases where an instruction is a critical producer for an instruction but a non-critical producer for another.

In our experiments with several encryption applications (AES-128 engine, SHA3, RSA, etc.), we found that an 8-bit offset is sufficient to represent the PCs of non-critical instructions in the DT because the PCs of instructions are not far from their dependent instruction PCs.

3.2 Criticality and Slack Detection

As explained in Section 2, slack is defined as the time difference between the production of an instruction’s operands.

Assuming the same example of Figure 1, the slack of instruction INST 2 will be the absolute difference of the times that its producers (INST 0 and INST 1) will return their result, represented by the Equation 1.

$$slack(INST\ 2) = abs(t_0 - t_1) \quad (1)$$

When a new instruction is issued, an entry is allocated for it in the Destination Table (DT). When the same instruction completes its execution, the Slack Unit detects the criticality it produces and calculates the slack. The critical instruction is stored immediately in the Critical Table (CT). At the same time the Non-Critical Table (NCT) is checked and if the current critical instruction matches with a previous NCT entry, the entry is dropped as the critical status supersedes. Before storing the non-critical instruction we must first query the CT for possible criticality conflicts. A criticality conflict occurs when an instruction is a producer for two or more instructions and its criticality status is different for each one of them. In this case the producing instruction is not stored in the NCT. In the absence of a conflict with the CT, the producer will be stored in the NCT table coupled with the calculated slack and marked as not stable. If the instruction was already stored in the NCT and the new slack is smaller than the previously stored one, we update the entry and add the new (smaller) slack. When an issued instruction hits the DT (indicating that it appeared before), we verify that the criticality status of its producers is the same and update the non-critical status in the NCT to stable. If their criticality status changed, we make the appropriate changes in all Slack Unit structures.

Although false hits in the Slack Unit can happen when context-switching processes with the same PCs, solutions exist (flushing the Slack Unit on context switches, or tagging tables with process-specific information), but the evaluation of these solutions fall outside the scope of this work.

3.3 Delay Injection

To find whether a newly dispatched instruction is to be reordered, the Slack Unit is queried (specifically the NCT) with its PC and in case of a hit, the appropriate delay is returned and injected to its issue slot. This delay is only applied just before the instruction is marked by the out-of-order scheduler as ready to execute. That is, after all its input operands have been produced. As soon as its operands are produced, the instruction will be marked to wait for another *delay* number of cycles. If the delay is marked in NCT as *stable*, it will be used as an upper bound to select a random value between 0 and the stable delay. If the delay is not stable, we label this delay as being in the *unstable phase*, and we don't randomize this delay. This is done to add an additional layer of randomization in the desynchronization of the execution. However, the delay will still be used as is, without randomization, if the slack is unstable to avoid adding unnecessary performance overheads in the execution. Note that the Slack Unit is constantly learning to determine the stable delay.

3.4 Example

Figure 4 shows an example of how we detect the criticality and slack of instructions and how we inject a delay on non-critical instructions to desynchronize the execution. When

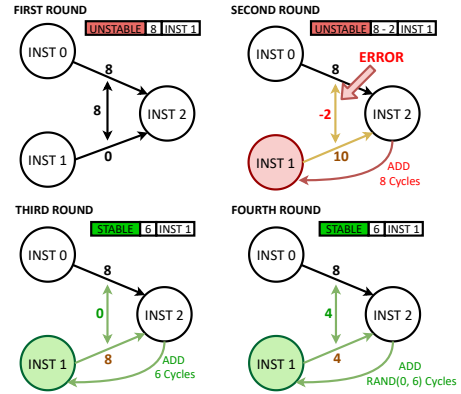


Figure 4: Slack detection and delay injection example.

INST 2 is issued in the first round, we detect the completion time for each of its input operands and using Equation 1 we calculate the slack (8 cycles). INST 0 is the critical instruction that we can't delay as it finished last and INST 1 is marked as non-critical with a slack of 8 cycles. Because INST 1 is still unstable, in its next appearance (second round) we inject it with the slack time as is. We expect that the new slack of INST 2 should now be 0 as INST 1 still needs 8 cycles to complete. However, practically we can find that INST 1 took 10 cycles to finish in this round (instead of the 8 cycles that we injected as a delay). The reason this happens is the backward dependencies of INST 1 to previous instructions that might have also been delayed. This is a common occurrence for all instructions that appear for the first time. This unstable delay slack injection occurs during the *unstable phase*.

During this unstable phase, the instruction is detected as unstable and the Slack Unit records the slack of INST 1 using Formula 2.

$$slack_{update}^i = slack_{old}^i - slack_{new}^i \quad (2)$$

The new slack of instruction i ($slack_{update}^i$) will be the difference between the old slack ($slack_{old}^i$) and the new ($slack_{new}^i$). When the unstable phase is over in the third round, i.e., the injected delay will not change the criticality of the consumer (INST 2 in Figure 4) anymore, the instruction will be marked as stable and the injected delay thereafter (fourth round) will be randomized with the updated slack being the upper bound of the random delay. This randomization introduces more noise into the power measurements, thus increasing the security of the system.

4. SECURITY EVALUATION METHODOLOGY

In this section, we provide a detailed description of the security evaluation used in this work. First, we briefly introduce the power analysis attack for the AES algorithm. Next, we discuss the dangers of using simple pass-fail security evaluation methods that many previous studies base their work upon. Finally, we describe the security framework and metrics used for analysis of our proposed design.

4.1 Power Analysis Attacks

One of the goals of a power analysis attack is to reveal the secret keys used in encryption algorithms by observing the power consumption of a processor. To accomplish this, the adversary collects a large set of power traces from the core during encryption processing to examine and detect data and power dependencies. In this section, we take a look at the AES-128 algorithm as an example of how to recover secret keys from a power trace.

In the first round of the AES-128 algorithm, a plaintext (in this example, a 16-byte input to the algorithm that will be encrypted) is loaded byte by byte and XOR-ed with the secret key to form the initial state of the ciphertext (the 16-byte encrypted output of the algorithm). Equation 3 shows the SubByte step that is applied to the initial state of the ciphertext, which is also called an Sbox operation.

$$I_n = Sbox[X_n \oplus K_n] \quad (3)$$

In Equation 3, I_n is the n^{th} byte of the intermediate value after the Sbox operation, X_n is the n^{th} byte of the plaintext, and K_n is the n^{th} byte of the secret key. The Sbox is a look-up table that takes a byte as input and substitutes it with another byte. For this attack, the Sbox is public and the plaintext (X_n) is known to the adversary.

The overall modus operandi of a typical power analysis attack targeting a small part of the key or subkey¹ is shown in Figure 5. First, the adversary needs to get a power model reflecting the power consumption behavior of the device. This can be done using a priori assumptions on leakage behavior like Hamming weight/distance model [12] (better known as unprofiled attack), or by trying to characterize the actual model (i.e. profiled attack), using, for example, Gaussian template or supervised machine learning. Using this model, the adversary only needs to try all 2^8 possible values for the secret key byte and examine if the computed intermediate value has a high statistical dependency with the collected power measurements. The attack outputs a probability for each of the 2^8 guesses to be correct, and is successful if the most likely one corresponds to the actual key. This effectively reduces the brute force attack complexity from 2^{128} to $2^{12} (16 \text{ bytes} \times 2^8)$ when using a divide and conquer methodology. Many different statistical distinguishers, or attack tools, have been introduced in literature, with Kocher’s Differential Power Analysis (DPA [28]), Correlation Power Analysis (CPA [12]), Mutual Information Analysis (MIA [9]) and the maximum likelihood template [15, 42] being some examples. In the rest of Section 4, and independently of which statistical distinguisher is used, we will refer to DPA as any attack taking advantage of varying plaintext. This includes, for example, the aforementioned DPA and CPA.

4.2 Limitations of Simple Pass-Fail Tests

T-test: The order of leakage is defined by the statistical moment in which the meaningful information depends. For example, first-order (respectively second-order) leakages extract information in the mean (respectively variance/co-variance). A first-order secure masking countermeasure ensures that no

¹Typically, for AES, the subkey is one byte, where each byte of the key is attacked independently.

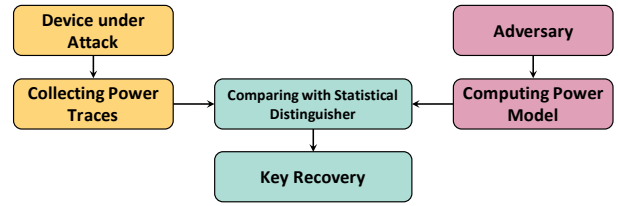


Figure 5: Flow of Differential Power Analysis (DPA) attacks. Power traces are collected from the device under attack and the adversary tries to compute the power model. A statistical distinguisher (like correlation, difference of means, etc.) is used to recover the secret key.

information lies in the mean of side-channel traces, forcing an adversary to extract information in (at least) the variance and co-variance. The Test Vector Leakage Assessment (TVLA [21]) methodology was introduced to detect the presence of this first-order leakage for masked implementations, and can be extended for higher orders [16]. That is, a T-test failure shows the existence of such leakage, while a success can only state that no such leakage was found for a given number of measurements. However, it has since often been used as a pass-fail method to evaluate the resilience of an implementation, where the hidden assumption is that a T-test failure with a total of Q measurements roughly translates to security up to Q measurements. However, such an assumption has been shown to be incorrect [45]. That is, the standard T-test should only be used to show the presence of first-order leakage when the null hypothesis is rejected. However, no conclusion should be made when it is not the case. As a direct consequence, using the T-test for the *hiding* style of countermeasure, where leakages would exist at first-order, should be avoided.

Fail/success unprofiled CPA: The strengths and weaknesses of unprofiled CPA lies in its assumptions. On one hand, it is designed to process one time sample at a time (also known as univariate leakage [45]). While one can combine time samples together before performing CPA (i.e. multivariate attack [45]), doing so is sub-optimal. On the other hand, CPA assumes some leakage model. For these reasons, CPA is often a good security estimate for unprotected implementations where the side-channel traces are well aligned and where the power model is well known (e.g. Hamming weight/distance). However, this no longer holds when the type of leakage diverge from these assumptions. That is, an implementation that changes the underlying leakage model (such as power balancing [24]), or that introduces jitter or misalignment from one measurement to another, will drastically drift apart from these hypotheses and subsequently makes CPA sub-optimal. In other words, *a failed CPA under sub-optimal assumptions does not provide a sound security evaluation*. A direct result of the use of this method will be a false sense of security (a higher-than expected security result), while a more appropriate attack (which exploits a well characterized leakage model or apply measurement alignment/processing techniques) might achieve key recovery with significantly fewer observations. In the next subsection, we propose an evaluation framework which also considers powerful adver-

sary capable of applying the right leakage model and processing techniques to exploit the available leakage in an optimal manner.

4.3 Proposed Evaluation Framework

The limitations of the previously discussed methods can be summarized as using weak or incorrect adversarial assumptions. As previously shown in Figure 5, the two main ingredients for a power attack are the leakage model and the statistical distinguisher used. The further the model is from the actual leakages from the device, the worse the attack will be. This issue typically happens when using a Hamming model for power balancing, for example, which inherently changes the leakage model. On the other hand, as methods such as CPA are not meant to combine leakages together, they are less optimal for multivariate attacks compared to template attacks [15]. As a result, evaluating the security of a device with the wrong model and method can lead to a false sense of security, while another method could succeed using fewer measurements. This can be mitigated by considering stronger adversarial capabilities, eventually up to a potentially non-existent (extremely strong) adversary in order to approach a lower-bound (conservative) estimate of security guarantees.

In this work, we will illustrate this by providing different levels of security analysis for three types of adversaries, that we call *basic*, *educated*, and *advanced*. First, in order to compare our work to existing literature, the basic evaluation considers an adversary that simply applies standard CPA with a Hamming weight model on the Sbox output [12]. Second, as a part of the security brought by our countermeasure comes from desynchronization, an educated evaluation will be performed with a CPA analysis with the same model, additionally pre-processing traces to defeat countermeasures with alignment techniques such as integrating the leakage over different time samples. This simple method aims to show how basic knowledge of the implementation and a simple attack twist can greatly change the evaluation outcome (effectively demonstrating a lower security guarantee). Finally, the advanced evaluation aims to approach the lower security bound (a conservative security estimate) by assuming an adversary adopting profiled attacks. That is, we will first take advantage of a profiling (or training) set in order to mount a multivariate template attack [15] augmented with profiled Principal Component Analysis (PCA) for dimensionality reduction [6]. PCA applies a linear transformation that projects high-dimensional data into a low-dimensional space while preserving the data variance, by computing the eigenvectors of the co-variance matrix. The training or profiling phase with PCA allows an adversary to learn the precise leakage model and better characterize the underlying countermeasure, leading to a confident lower-bound on security.

4.4 Security Metrics

As the divide-and-conquer approach allows one to attack each byte of the key independently, we target an attack with only one key byte without loss of generality. For all three attack methodologies, the resulting vector of 256 probabilities/scores for each key guess is denoted by \mathbf{p} . In our evaluation, we will compute the security using the following metrics:

Table 1: System configuration.

Parameter	Value	Parameter	Value
RF/Fetch Buffer/ROB	128/24/96 entries	L2 Cache	4 MB, 8-way set assoc.
Issue Queue	3×8 entries	Bus Protocol	AXI
Execution Units	5 (1 MEM, 3 ALUs, 1 FPU)	*(PARADISE) DT	208 B, 4-way set assoc.
Branch Predictor	Next-line, backing predictor	*(PARADISE) CT	144 B, 4-way set assoc.
Cache line size	64 B	*(PARADISE) NCT	192 B, 4-way set assoc.
L1-I and L1-D Caches	32 KB, 8-way set assoc.		

Key rank (byte): Given the probability/score vector \mathbf{p} resulting from an attack, the rank of the key (byte) is given by the number of key candidates having a higher probability than the correct key.

Guessing entropy [46]: For a given key byte k , the guessing entropy (GE) is the average key byte rank within its vector of probability \mathbf{p} . We define by $\text{rank}(\mathbf{p}, k)$ the function that returns the rank of the subkey k within the vector \mathbf{p} . From a set of n_a independent attack result vectors \mathbf{p}_i , Equation 4 allows one to compute the guessing entropy.

$$\text{GE} = \frac{\sum_{i=0}^{n_a-1} \text{rank}(\mathbf{p}_i, k)}{n_a}. \quad (4)$$

The use of guessing entropy provides more information than the commonly used measurement to disclosure (MtD) metric [30]. First, it provides averaged information over several independent attacks. This minimizes the over- and under-estimation of the actual security, as a single experiment could be an outlying result. Second, it additionally shows the global key recovery progression instead of only reporting the overall number of traces. Indeed, as these attacks belong to the class of divide-and-conquer methodologies, one can trade-off side-channel complexity for a computational one and recover the key through enumeration before the rank reaches one [38]. As a result, only looking at the number of measurements required to recover the key can be misleading, as the implementation might be broken with fewer traces with some brute-force or key enumeration.

5. EXPERIMENTAL SETUP

We implement PARADISE on top of SonicBOOM [51], an open-source RISC-V out-of-order processor. The configuration of the processor is shown in Table 1. We leverage Galois Linear Feedback Shift Register (GaloisLFSR) function [31, 39] provided by Chisel [8] to randomize the stable delay. The system time when we generated the core and the runtime clock cycles are used together to generate a random seed. This random seed is more secure than the default seed (only runtime clock cycles) as it is almost impossible for the adversary to know when the core was generated.

Apart from the baseline (SonicBoom) and PARADISE, in this paper, we also implement three generic processors to compare with the PARADISE implementation. For each of these processors, we naively inject random delays (up to 8 cycles) by a given probability to match either the performance of our PARADISE implementation (in the random-iso-performance configuration), or the level of security protection of our implementation (in the random-iso-security design). The detailed configuration parameters of these processors and all other processors that are used in security and performance

Table 2: Implementation of different processors.

Processor Name	Platform	Description
ooo-baseline	SonicBoom	Unprotected baseline out-of-order processor
io-baseline	Rocket chip	Unprotected in-order processor
PARADISE	SonicBoom + Slack Unit	Secure instruction scheduling processor
random-iso-perf	SonicBoom + random delay	Delay injection probability is 5% Try to match the performance of PARADISE
random-iso-security	SonicBoom + random delay	Delay injection probability is 20% Try to match the advanced security evaluation of PARADISE
random-aggressive	SonicBoom + random delay	Inject random delay for all instructions Naive and aggressive implementation

evaluation are described in Table 1 and Table 2.

Benchmarks. To determine the exact performance degradation for the encryption application used in this study, we run the AES-128 encryption engine with 2,000 plaintexts both on SonicBoom and PARADISE. We then run the micro-benchmarks provided by Chipyard [5] on the out-of-order cores from Table 2 to evaluate the overheads as introduced by our protection scheme and the random-delay injection schemes for a set of general-purpose applications. The micro-benchmarks consist of a basic set of applications designed to test the functionality of the processor in different scenarios, e.g., complex matrix computation, multi-threaded applications, sorting, Dhrystone, etc. Finally, we boot a full Linux system on each core using FireSim [27] to run the the SPEC CPU2017 benchmark suite. This allows us to observe the performance of different processors in a more general environment. However, because of limitations in FireSim, we support only 10 SPEC CPU2017 benchmarks.

Area and Power. We leverage the Synopsys Design Compiler (DC) to synthesize the PARADISE and the SonicBoom, both of which are generated by the default synthesizable SonicBoom configuration. Next, we use VCS to perform the gate-level simulation on synthesized processors to generate realistic gate activity. PrimePower is then used to generate the power consumption of the processor by analyzing the gate-level waveform and gate-level code.

Power traces simulation framework. We use a Hamming weight leakage model to generate power traces. Equation 5 shows the estimated power at any given time of the execution.

$$\text{power}(T) = \begin{cases} 0 & \nexists \text{ inst} : \text{inst.WB} = T \\ \sum_{\text{inst}} \text{HW}(\text{inst}) & \forall \text{ inst} : \text{inst.WB} = T \end{cases} \quad (5)$$

The write-back time of the instruction *inst* to the register file is denoted by *inst.WB*. Also, *HW(inst)* is the Hamming weight of the data that instruction *inst* writes to the register file. We use the behavioral simulation of the SonicBoom core to gather this information and generate the power traces. We only use the updates to the register file to generate the power traces since the register file in SonicBoom consumes much more power than system and memory bus (33× more power consumption for integer register file compared to system and memory bus, and 61× more power consumption for integer+floating-point RF).

To perform security evaluations, each implementation consists of two sets of one million traces each. The first set (attack set), is composed of a fixed key and randomly varying plaintext. The second set (profiling set), is composed of randomly varying keys and plaintext that are known by the

adversary to perform advanced profiling methods.

6. EXPERIMENTAL RESULTS

6.1 Security Evaluation

To compare our work to the literature, we perform a basic evaluation, which performs a regular CPA without any modifications. Next, we assume an adversary that knows the desynchronization aspect of our countermeasure in order to perform the educated evaluation. The adversary will now use integration over the time samples to reduce the signal reduction coming from the desynchronization. This will show that basic insight and twist from the adversary can greatly change the security evaluation outcome and give a false sense of security. Finally, we present results of our advanced evaluation that aims to approach the lower security bound by assuming a strong profiled adversary. For each method we compare the security benefits of all cores with their corresponding ooo-baseline with respect to the same attack.

6.1.1 Basic evaluation

As a first analysis, we divided our 10 million attack traces into 20 subsets, and performed a standard CPA on each of them and average them to compute the guessing entropy. The results are shown in Figure 6, and will be the main point of comparison with other works. The x-axis corresponds to the number of traces and the y-axis corresponds to guessing entropy. A guessing entropy of 0 indicates that the correct key is highest ranked (on average) and thus the attack is successful.

As a first observation, we can observe the gap between io-baseline and ooo-baseline sets, which respectively recovers the key with 500 and 1,800 traces. This shows the simple security benefit of using an out-of-order core instead of in-order ones. Indeed, out-of-order core still provides some randomness in the computation timings with a small security benefit. As ooo-baseline corresponds to the unprotected implementation on our out-of-order case study, we will use it as a reference for our security benefits. First, we can see that the PARADISE version of our countermeasure increases the number of traces needed to 470,000 when using standard CPA. When considering basic evaluation, this shows a security of $261 \times$ ooo-baseline. However, using random delays with the same performance as PARADISE, shown by the random-iso-perf performances, only requires 22,000 traces for key recovery, only corresponding to a benefit of $12 \times$. This shows that, when considering standard CPA, our method shows greater security benefits than random delays with overheads. Finally, random-aggressive only requires 220,000 traces, which corresponds to a security benefit of $122 \times$, and is less than the PARADISE implementation. While this can look surprising at first, it can be explained when looking at the two security benefits brought by our countermeasure. As explained in Section 2, the security mainly comes from (1) desynchronization and (2) more randomness in the register's content. The random-aggressive implementation focuses on increasing the desynchronization, without much change in the register content randomization, which is higher in the case of the PARADISE one.

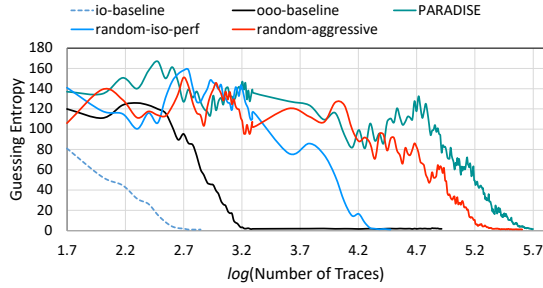


Figure 6: Results of the basic security evaluation. The x-axis corresponds to the number of traces, and the y-axis corresponds to guessing entropy.

6.1.2 Educated evaluation

As a second evaluation, we illustrate how some basic knowledge on the type of implemented countermeasure, and a simple optimization of the attack itself can drastically change the outcome. As a big part of security brought by our countermeasure is brought by desynchronization, standard CPA being a univariate attack is inherently suboptimal. Instead, we now assume an adversary with some insight that simply combines sets of N consecutive time samples on the trace together (time integration) prior to performing the CPA to reduce the effect of desynchronization. For each implementation, we tested values of $N = 20, 50, 100, 150, 200$, for which the best corresponding results are shown in Figure 7. Again, the x-axis corresponds to the number of traces and the y-axis represents guessing entropy.

We do not report any results for io-baseline and ooo-baseline implementations, as basic evaluation was better (in term of number of traces required) for non-existent and limited desynchronization in io-baseline and ooo-baseline respectively. For that reason, we will still use the numbers from the basic evaluation for these two implementations. However, we can see that the educated evaluation drastically reduces the required number of traces for both PARADISE and random-aggressive implementations, which now respectively broken with 22,250 ($12\times$) and 20,000 ($11\times$) traces instead of 470,000 ($261\times$) and 220,000 ($122\times$) respectively. This simple attack optimization shows the danger of using sub-optimal attack strategies for security evaluations. Interestingly, minor benefits are seen for random-iso-perf implementation where 21,000 traces are now required instead of 22,000.

6.1.3 Advanced evaluation

Our last evaluation considers a powerful adversary being able to profile the leakages using, for example, a copy or clone of the device under attack for which she has complete control. First, we use profiled CPA [19] in order to identify leaking features in the trace. The results are shown in Figure 8, where the left part shows the results for the ooo-baseline and the right part shows results for random-aggressive implementations. The x-axis corresponds to the time samples, while the y-axis shows the correlation coefficient.

As we can see, leakages are clearly identified with several peaks for the ooo-baseline implementation. We observed

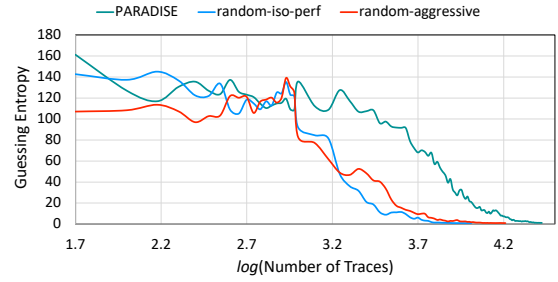


Figure 7: Results of the educated security evaluation. The x-axis corresponds to the number of traces, and the y-axis corresponds to guessing entropy.

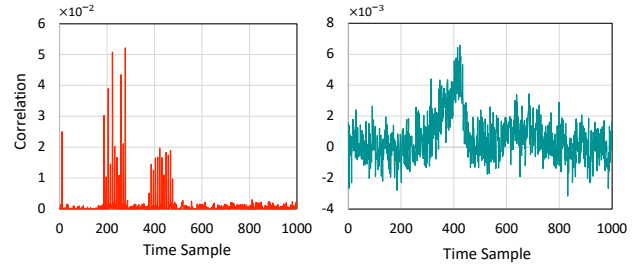


Figure 8: Leaking regions using profiled CPA for ooo-baseline (left) and random-aggressive (right). The x-axis corresponds to the time samples, and the y-axis corresponds to correlation coefficient.

similar behavior for the io-baseline and random-iso-perf one. For these two implementations, we thus selected all time samples having a correlation above 0.005 as valid attack samples. However, the leaking samples for the random-aggressive implementation are less clearly identified, as shown by the Gaussian shape correlation trace covering around 200 time samples. This was similarly observed for PARADISE and random-iso-perf implementations, which is the result of the desynchronization brought by the countermeasures. In that case, we selected all time samples happening before and after the peak regions as valid.

Once the points of interest are selected, we perform a profiled Principal Component Analysis (PCA) [6] in order to further reduce the dimensionality and to project the sample into a more informative space. The result of the projection is then fed into a multivariate template attack [15]. For each implementation, we use a different number of principal components, and show the best results for each of them in Figure 9. We additionally show the results for the random-iso-security implementation, having similar security performances as the PARADISE one (tailored specifically in the case of the advanced evaluation).

We can see that the advanced method produces better results in term of attack power than the basic and educated ones. First, io-baseline implementation now only requires 125 traces instead of 500 when using standard CPA. Second, ooo-baseline implementation is now broken with only 400

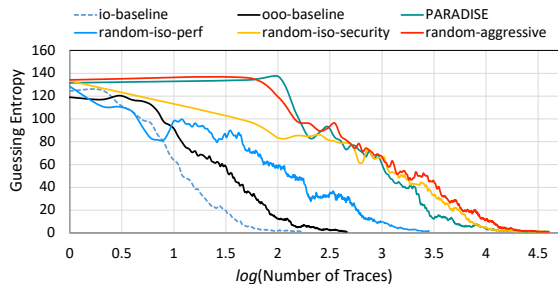


Figure 9: Results of the advanced security evaluation. The x-axis corresponds to the number of traces, and the y-axis corresponds to guessing entropy.

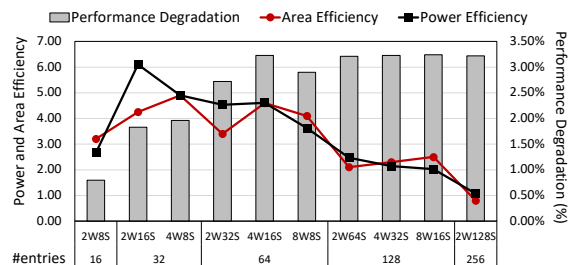


Figure 10: Performance degradation compared to ooo-baseline core and area/power efficiency using different parameters. W and S represent number of ways and sets respectively, the #entries are equal to $W \times S$, which denote the total number of entries of each table in the Slack Unit.

traces, which we will now use as comparison reference as opposed to 1,800 for basic evaluation. Next, PARADISE implementation now only requires 13,500 traces, showing a security gain of $34\times$. However, random-iso-perf is now broken with 2,200 traces, hinting that our countermeasure is $5.5\times$ more secure than random delays when considering a strong adversary. Interestingly, as opposed to previous results, the random-aggressive implementation now requires 15,000 traces, which is more than the PARADISE one with a benefit of $38\times$. Indeed, as we now profile the leakage model, the effect of the vertical noise is now reduced, which has more impact on the PARADISE implementation. Overall, this shows that using the wrong or too sub-optimal attack against a given countermeasure can lead to a false sense of security. Indeed, from standard CPA to multivariate templates, the number of required traces has been divided by 35 for the PARADISE implementations, and by 15 for the random-aggressive one due to wrong model assumptions. However, the number of traces unprotected ooo-baseline implementation was only divided by 4.5, as the model was already fitting more. From these observations, we highlight as a cautionary note when presenting results from two different implementations, that it can only reflect the adversary’s assumptions. On that matter, it is more conservative to apply powerful attacks, thus assuming a strong adversary.

6.2 Performance Evaluation

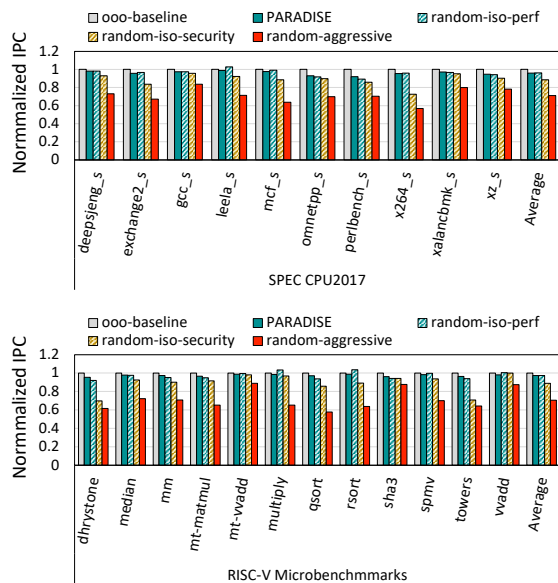


Figure 11: Performance of different cores normalized to ooo-baseline core. *We run all SPEC CPU2017 benchmarks compatible with FireSim.

We evaluated several configurations of PARADISE by running an AES-128 encryption engine with 2,000 plaintexts to find the best combination of performance, power and area. Figure 10 shows the best combination of power and area efficiency to be a 4-way and 16-set Slack Unit. Power-performance and area-performance efficiency shown in Figure 10 are calculated as $\frac{\text{overhead}_{\text{perf}}}{\text{overhead}_{\text{power}}}$ and $\frac{\text{overhead}_{\text{perf}}}{\text{overhead}_{\text{area}}}$. We use the highest number for $\text{overhead}_{\text{perf}}$ because in this scenario PARADISE is going through a longer *unstable phase* and although it injects more unstable delays, it collects more runtime information to improve in later iterations. Therefore, more instructions will be reordered and the desynchronization of the execution increases with the overall security improved. The rest of our performance results were taken with all structures in the Slack Unit being 4-way with 16 sets.

Figure 11 shows the performance of PARADISE (normalized IPC to the ooo-baseline) on two different sets of benchmarks. At the bottom, the RISC-V bare-metal microbenchmarks show an average overhead on PARADISE of 2.6% (maximum is 4.8 %). For the random-iso-perf, random-iso-security, and random-aggressive implementations we get 2.7%, 11%, and 28% overheads respectively. However, these applications don’t have a real software stack and some of them cannot generate a stable result due to the limited number of instructions. For this reason we evaluated a subset of SPEC CPU2017 benchmarks (top of Figure 11). We could only evaluate those applications that were compatible with FireSim. The average overhead of PARADISE in this case is 4%, while for random-iso-perf, random-iso-security, and random-aggressive it is 4.2%, 12%, and 29% respectively. For all the applications evaluated, the average performance overhead of PARADISE is 3.7%. An in-depth analysis of the performance results showed that the overhead produced comes mostly from the *unstable phase* where we inject an

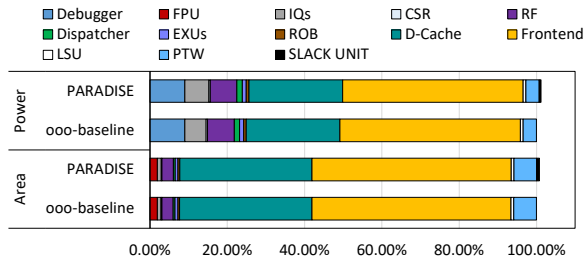


Figure 12: Power and area overheads of PARADISE.

unstable delay. As soon as the criticality of the instructions and the slack become stable, after a few iterations of the algorithm, the performance of the processor returns to its baseline levels.

6.3 Power and Area Overheads

Figure 12 shows the power and area of PARADISE compared to the ooo-baseline. The Slack Unit and the GaloisLFSRs (for randomizing the delay) introduces a negligible overhead of 1.1% on PARADISE over the baseline core. In matters of area, the total overhead of PARADISE compared to the ooo-baseline is 0.7% that comes from the Slack Unit, the and GaloisLFSRs, the delay controller on every issue slot and the wire connections for all the new components.

7. RELATED WORK

Table 3 summarizes state-of-the-art countermeasures, and compares their reported performance, power, area, security evaluation and other implementation-based features.

7.1 Power Analysis Attack Countermeasures

Obfuscated execution. One solution to overcome power analysis attacks is to obfuscate the execution and operate on the data in an obfuscated form [32]. A solution for encryption engines implemented on a CGRA is proposed in ARDPE [20]. To make power and data correlation more difficult between different encryption rounds, the register usage of instructions and the data going to the encryption engine is randomized. Although ARDPE is one of the lowest overhead techniques (less than 10% performance and area overhead), it specifically targets encryption engines that take plaintexts as input. Compiler support and more hardware structures will be required to apply this technique for general-purpose processors. More generic solutions have been proposed with [1, 7] aiming to secure general-purpose RISC-V in-order cores. In [1], the authors propose a hardware-software co-operative technique to detect the leakiest moments during encryption, and using a software controller to electrically disconnect the core from the system. During the disconnection, the power traces will show no correlation with the data that is being processed, however this technique can increase execution time by as much as $2.7\times$. PARAM [7] is another solution that investigates the leakiest modules of a RISC-V design and addresses them separately with appropriate countermeasures. A part of the leakages they work on are the translations in EDA tools (like Bluespec compiler [35]), hence they modify the RTL code to prevent these leakages. They also address the leakages of the data in the register file and buffers by ob-

fuscating and de-obfuscating the data upon each access. In other words, they always keep the data in the obfuscated form and de-obfuscate it when they need to process the data. The authors do not report the performance impacts of their countermeasures for the leakage they address.

Random code injection. In RIJID [3], the authors propose a hardware-software co-design that inserts random (irrelevant) instructions in the execution at random intervals. They use the compiler to detect the regions of the code that need protection, and the hardware injects random instructions during the execution of the specified regions. Inserting real instructions in the processor during normal execution results in increased execution times. Specifically in RIJID, the performance overhead is 30% over their non-secure baseline. Others have adopted similar approaches with code injection with similar outcomes [2, 4].

Random instruction shuffling. In Block Shuffler [10], the authors propose shuffling independent blocks of instructions in a random order. In their hardware-software co-operative design, the compiler detects the independent blocks of the code and inserts `shuffle` instructions that allow the *shuffler unit* in an in-order processor to generate a random permutation of the order of instruction blocks and start fetching instructions based on that order. The reported overhead in performance, power and area is negligible. The performance overhead is low because different rounds of an encryption algorithm can execute independently in any order without any impact on the control flow and data flow. The processor only requires the knowledge from the compiler. The granularity of the shuffling in this technique is an instruction block. In PARADISE, we shuffle the instructions in a finer grained way and take advantage of the slack for each instruction. These two techniques are orthogonal and can be combined. However, the block shuffling technique requires re-compilation and modifications in the code in order to inform the processor which instruction blocks are independent and can be re-ordered. Also, this technique might not be as effective in out-of-order cores, since these cores re-order the execution of instructions for performance reasons and could undo parts of the effects of block shuffling, especially when the instruction blocks are much smaller than the instruction window. There are also hardware- and algorithm-specific shuffling techniques to overcome power analysis attacks [36].

Circuit-level protection. Power balancing techniques try to minimize the side-channel leakage by balancing the power consumption of the core at all times of the execution [13, 24, 34, 44, 47]. Previous work implements *gate-level masking* that uses complex logic gates [37, 41]. However, existing power balancing and masking techniques are not generic for all encryption algorithms, and they have high overheads for performance, power, and area. Other works [50] use a lightweight technique that combines power balancing and hardware masking is proposed. However, it is specific for AES engines that use a fixed key. Noise injection and power isolation are other techniques that hide the intermediate values of encryption operations [17, 22, 25, 26, 48, 49]. The authors of ANSI [17] combine these two techniques to implement a generic solution for encryption algorithms with negligible performance overhead. In summary, each of the generic solutions evaluated in this work demonstrate high

Table 3: Comparison of existing power attack countermeasures. *For each method we compare the security benefits of each core to their corresponding unprotected baseline with respect to the same attack. Values of other works are presented as reported; '-' for imprecise reporting or absence of reporting of the information. ‡ Evaluation reported as negative and/or inconclusive tests that did not recover the key.

Paper	Hardware Agnostic	Algorithm Agnostic	No Re-compile	Design	Area	Overheads*		Security Evaluation*			Technique
						Power	Performance	Basic	Educated	Advanced	
WDDL [24]			✓	VLSI	200%	300%	300%	128×	-	-	Power Balancing
IVR [25]		✓	✓	VLSI	100%	100%	0%	‡	-	-	Voltage Regulation
False-Key [50]			✓	VLSI	3%	0%	2%	187×	-	-	Gate-Level Masking
ASNI [17]		✓	✓	VLSI	60%	68%	0%	1000×	1000×	-	Noise Injection
Blinking [1]	✓	✓	✓	VLSI/SW	-	-	270%	10 – 100×	10 – 100×	-	Power Hiding
PARAM [7]	✓	✓	✓	µarch	~20%	-	-	‡	-	-	Data Obfuscation
ARDPE [20]		✓	✓	µarch	7.23%	-	3.4%	4000×	-	-	Data Randomization
RIJID [3]	✓	✓		µarch/SW	2%	27%	30%	‡	-	-	Random Code Injection
Block Shuffler [10]	✓	✓		µarch/SW	2%	1.5%	0.7%	‡	-	-	Coarse Instr. Shuffling
random-iso-perf	✓	✓	✓	µarch	~0%	0.8%	3.8%	12×	12×	5.5×	Fine Instr. Re-ordering
random-iso-security	✓	✓	✓	µarch	~0%	0.4%	11%	-	-	34×	Fine Instr. Re-ordering
random-aggressive	✓	✓	✓	µarch	~0%	0%	29%	122×	11×	38×	Fine Instr. Re-ordering
PARADISE (this work)	✓	✓	✓	µarch	0.7%	1.1%	3.7%	261×	12×	34×	Fine Instr. Re-ordering

power and/or area overheads. In addition they are designed for encryption-specific hardware only.

7.2 Security Evaluation

As discussed in Section 4, we consider three types of adversaries for security evaluation: (1) a basic evaluation where the adversary is given a completely wrong model which gives a false sense of security, (2) an educated evaluation where the given security model fits the type of countermeasure, and (3) an advanced evaluation where the adversary extracts the actual leakage model. Our investigations show that none of the papers in Table 3 perform the advanced (profiling/multivariate/worst-case) analysis and most of the reported analyses fall into the basic category.

WDDL [24] evaluates the effect of their technique using basic CPA with Hamming model (HW for the unprotected and HD model for protected devices) and claim 128× security increase. However, using the Hamming model is inherently flawed for power-balancing-like countermeasures, as now the leakage does not correspond to the bit switch value anymore [11]. For this countermeasure, the adversary has to model the function $f(x)$ that shows the consumption difference of $x - \bar{x}$. This can be done using non-profiled linear regression [42], or profiled attacks [15]. As a result, the evaluation of this paper falls into the basic evaluation. The False-Key [50] paper also suffers from the same issue as the WDDL paper, since they use a Hamming model. Their analysis is performed using basic CPA in two stages: (1) targeting the false key at the Sbox output, and (2) targeting the actual key during the re-computation phase which uses power balancing. The security benefits are only brought by the second phase. Moreover, they claim their countermeasure is a masking technique, but all the leakages are first-order from a statistical point of view. They do not take into account that an advanced adversary can reverse-engineer the fixed pre-computed Sboxes and recover the secret key this way. Hence, the security evaluation in this paper falls into the basic category.

The security evaluation of the IVR [25] paper also falls into the basic category. They use standard CPA and T-tests [16] and report a 20× security increase. Since their countermea-

sure adds jitter, using basic univariate methods cannot be very effective in recovering the secret key and it can be improved by using methods like simple averaging. Moreover, the tests on the protected implementation are negative (inconclusive) which do not allow one to properly quantify the security improvement. For example, rank estimation methods [38] could have been used to provide an improved security trend.

In PARAM [7], basic CPA is used for security evaluation. They report that they break the secret key within 60K measurements, while the protected implementation is resistant with up to 1 million measurements. Due to the nature of their countermeasure, evaluating the security with basic CPA is good to see the effect of the presence of non-obfuscated leakage. However, as explained for IVR, negative (inconclusive) results are not sufficient to conclude security. Moreover, the security of the obfuscation itself is not investigated. Therefore, we classify the evaluation of PARAM as basic. ARDPE [20] paper also uses basic CPA for evaluation and it falls into the basic evaluation category. They show 4000× security increase compared to the unprotected implementation. However, their countermeasure adds desynchronization and using basic CPA is a weak evaluation method which provides a false sense of security, since an advanced adversary can reveal the key with fewer measurements.

The security evaluations of ANSI [17] and Blinking [1] papers fall into the educated category. ANSI uses basic CPA and reports 1000× security increase. Since the effect of their countermeasure is basically an SNR reduction, without any changes on the power model, evaluating the security using basic CPA is fairly adequate. The Blinking paper uses the T-test and the sum of mutual information over all samples, which shows a security increase of 10×-100× for different implementations. They are also reducing SNR without any changes in the power model. Hence, their security evaluation is considered to be the educated model, just as the ANSI paper. The authors of the RIJID [3] and Block Shuffler [10] papers do not mount actual attacks to recover the secret keys. For example in Block Shuffler [10], 100K traces are used to attack the protected core with basic DPA attack and they are not able to recover the key. But the authors do not provide the actual number of traces required for a successful attack.

8. CONCLUSION

In this work we take on the challenge of designing an efficient, secure and general-purpose processor that can protect all executing applications against side channel attacks, without affecting their performance. We propose a secure, fine-grained scheduling algorithm that dynamically reorders non-critical instructions in a random but calculated manner to desynchronize the execution and create non-deterministic behavior that increases the measurement noise. To achieve this, we exploit the time between operand availability of critical instructions (*slack*) to create high-performance random schedules. In addition, we provide a comprehensive security evaluation model that includes three security evaluation standards to demonstrate more robust attacks. Our proposed solution, PARADISE, offers a stronger security guarantee than demonstrated in previous works, even when tested on a more advanced and realistic security evaluation that complies with the highest security standards. PARADISE improves security against power analysis attacks by $34\times$ to $261\times$ with power and area overheads of 1.1% and 0.7% respectively. Moreover, our system achieves performance within 96%, on average, of the baseline unprotected processor.

REFERENCES

- [1] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. Hiding intermittent information leakage with architectural support for blinking. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 638–649. IEEE, 2018.
- [2] Jude A. Ambrose, Roshan G. Ragel, and Sri Parameswaran. Randomized instruction injection to counter power analysis attacks. *ACM Trans. Embed. Comput. Syst.*, 11(3), September 2012.
- [3] Jude Angelo Ambrose, Roshan G. Ragel, and Sri Parameswaran. Rijid: Random code injection to mask power analysis based side channel attacks. In *Proceedings of the 44th Annual Design Automation Conference, DAC '07*, page 489–492, New York, NY, USA, 2007. Association for Computing Machinery.
- [4] Jude Angelo Ambrose, Roshan G. Ragel, and Sri Parameswaran. A smart random code injection to mask power analysis based side channel attacks. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '07*, page 51–56, New York, NY, USA, 2007. Association for Computing Machinery.
- [5] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.
- [6] Cédric Archambeau, Eric Peeters, F-X Standaert, and J-J Quisquater. Template attacks in principal subspaces. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–14. Springer, 2006.
- [7] M. Arsath K F, V. Ganesan, R. Bodduna, and C. Rebeiro. Param: A microprocessor hardened for power side-channel attack resistance. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 23–34, 2020.
- [8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.
- [9] Lejla Batina, Benedikt Gierlichs, Emmanuel Prouff, Matthieu Rivain, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Mutual information analysis: a comprehensive study. *Journal of Cryptology*, 24(2):269–291, 2011.
- [10] Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne, and Wayne Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–19, 2012.
- [11] Shivam Bhasin, Sylvain Guilley, Florent Flament, Nidhal Selmane, and Jean-Luc Danger. Countering early evaluation: an approach towards robust dual-rail precharge logic. In *Proceedings of the 5th Workshop on Embedded Systems Security*, pages 1–8, 2010.
- [12] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.
- [13] Marco Bucci, Luca Giancane, Raimondo Luzzi, and Alessandro Trifiletti. Three-phase dual-rail pre-charge logic. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 232–241. Springer, 2006.
- [14] Robert Callan, Alenka Zajić, and Milos Prvulovic. A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47*, pages 242–254, Washington, DC, USA, 2014. IEEE Computer Society.
- [15] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.
- [16] Jeremy Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Pankaj Rohatgi, et al. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference*, volume 20, 2013.
- [17] D. Das, S. Maity, S. B. Nasir, S. Ghosh, A. Raychowdhury, and S. Sen. Asni: Attenuated signature noise injection for low-overhead power side-channel attack immunity. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(10):3300–3311, 2018.
- [18] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. Side-channel vulnerability factor: A metric for measuring information leakage. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 106–117, June 2012.
- [19] François Durvaux and François-Xavier Standaert. From improved leakage detection to the detection of points of interests in leakage traces. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 240–262. Springer, 2016.
- [20] Wei GE, Shenghua CHEN, Benyu LIU, Min ZHU, and Bo LIU. A power analysis attack countermeasure based on random data path execution for cgra. *IEICE TRANSACTIONS on Information and Systems*, 103(5):1013–1022, 2020.
- [21] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*, volume 7, pages 115–136, 2011.
- [22] Tim Güneysu and Amir Moradi. Generic side-channel countermeasures for reconfigurable devices. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 33–48. Springer, 2011.
- [23] Zhangqing He, Tianyong Ao, Meilin Wan, Kui Dai, and Xuecheng Zou. Erist: An efficient randomized instruction insertion technique to counter side-channel attacks. *IAENG International Journal of Computer Science*, 43(1), 2016.
- [24] David D Hwang, Kris Tiri, Alireza Hodjat, B-C Lai, Shenglin Yang, Patrick Schaumont, and Ingrid Verbauwhede. Aes-based security coprocessor ic in 0.18-*mu*hboxm cmos with resistance to differential power analysis side-channel attacks. *IEEE Journal of Solid-State Circuits*, 41(4):781–792, 2006.
- [25] Monodeep Kar, Arvind Singh, Sanu Mathew, Anand Rajan, Vivek De, and Saibal Mukhopadhyay. Improved power-side-channel-attack resistance of an aes-128 core via a security-aware integrated buck voltage regulator. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 142–143. IEEE, 2017.
- [26] Monodeep Kar, Arvind Singh, Anand Rajan, Vivek De, and Saibal Mukhopadhyay. An integrated inductive vr with a 250mhz all-digital multisampled compensator and on-chip auto-tuning of coefficients in 130nm cmos. In *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, pages 453–456. IEEE, 2016.
- [27] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin

- Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 29–42, Piscataway, NJ, USA, 2018. IEEE Press.
- [28] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.
- [29] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 87–101, New York, NY, USA, 2015. Association for Computing Machinery.
- [30] Stefan Mangard. Hardware countermeasures against dpa—a statistical analysis of their effectiveness. In *Cryptographers' Track at the RSA Conference*, pages 222–235. Springer, 2004.
- [31] George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.
- [32] David May, Henk L. Muller, and Nigel P. Smart. Non-deterministic processors. In Vijay Varadharajan and Yi Mu, editors, *Information Security and Privacy*, pages 115–129, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [33] Daniel S. McFarlin, Charles Tucker, and Craig Zilles. Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism? In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 241–252, New York, NY, USA, 2013. Association for Computing Machinery.
- [34] Maxime Nassar, Shivam Bhasin, Jean-Luc Danger, Guillaume Duc, and Sylvain Guilley. Bcd1: A high speed balanced dpl for fpga with global precharge and no early evaluation. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 849–854. IEEE, 2010.
- [35] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.
- [36] S. Patranabis, D. B. Roy, P. K. Vadnala, D. Mukhopadhyay, and S. Ghosh. Shuffling across rounds: A lightweight strategy to counter side-channel attacks. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 440–443, 2016.
- [37] Thomas Popp, Mario Kirschaum, Thomas Zefferer, and Stefan Mangard. Evaluation of the masked logic style mdpl on a prototype chip. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 81–94. Springer, 2007.
- [38] Romain Poussier, François-Xavier Standaert, and Vincent Grosso. Simple key enumeration (and rank estimation) using histograms: an integrated approach. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 61–81. Springer, 2016.
- [39] William H Press, H William, Saul A Teukolsky, A Saul, William T Vetterling, and Brian P Flannery. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.
- [40] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., August 2015. USENIX Association.
- [41] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *Annual Cryptology Conference*, pages 764–783. Springer, 2015.
- [42] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 30–46. Springer, 2005.
- [43] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [44] Danil Sokolov, Julian Murphy, Alexander Bystrov, and Alexandre Yakovlev. Design and analysis of dual-rail circuits for security applications. *IEEE Transactions on Computers*, 54(4):449–460, 2005.
- [45] François-Xavier Standaert. How (not) to use Welch's t-test in side-channel security evaluations. In *International Conference on Smart Card Research and Advanced Applications*, pages 65–79. Springer, 2018.
- [46] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 443–461. Springer, 2009.
- [47] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th European solid-state circuits conference*, pages 403–406. IEEE, 2002.
- [48] Carlos Tokunaga and David Blaauw. Secure AES engine with a local switched-capacitor current equalizer. In *2009 IEEE International Solid-State Circuits Conference-Digest of Technical Papers*, pages 64–65. IEEE, 2009.
- [49] C. Wang, M. Yan, Y. Cai, Q. Zhou, and J. Yang. Power profile equalizer: A lightweight countermeasure against side-channel attack. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 305–312, 2017.
- [50] W. Yu and S. Köse. A lightweight masked AES implementation for securing IoT against CPA attacks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(11):2934–2944, 2017.
- [51] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd generation Berkeley out-of-order machine. *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.