PARADISE: Criticality-Aware Instruction Reordering for Power Attack Resistance

YUN CHEN*, National University of Singapore, Singapore
ALI HAJIABADI*, National University of Singapore, Singapore
ROMAIN POUSSIER, Nanyang Technological University, Singapore
YASWANTH TAVVA, National University of Singapore, Singapore
ANDREAS DIAVASTOS, National University of Singapore, Singapore
SHIVAM BHASIN, Temasek Laboratories @ Nanyang Technological University, Singapore
TREVOR E. CARLSON, National University of Singapore, Singapore

Power side-channel attacks exploit the correlation of power consumption with the instructions and data being processed to extract secrets from a device (e.g., cryptographic keys). Prior work primarily focused on protecting small embedded micro-controllers and in-order processors rather than high-performance, out-of-order desktop and server CPUs. In this paper, we present PARADISE, a general-purpose out-of-order processor with always-on protection, that implements a novel dynamic instruction scheduler to provide obfuscated execution and mitigate power analysis attacks. To achieve this, we exploit the time between operand availability of critical instructions (*slack*) and create high-performance random schedules.

Further, we highlight the dangers of using incorrect adversarial assumptions, which can often lead to a false sense of security. Therefore, we perform an extended security analysis on AES-128 using different levels of adversaries, from basic to advanced, including a CNN-based attack. Our advanced security evaluation assumes a strong adversary with full knowledge of the countermeasure and demonstrates a significant security improvement of 556× when combined with Boolean Masking over a baseline only protected by masking, and 62,500× over an unprotected baseline. The resulting overhead in performance, power and area of PARADISE is 3.2%, 1.2% and 0.8% respectively[1].

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**; • **Computer systems organization →** *Architectures*.

Additional Key Words and Phrases: Power Analysis Attacks, Side-Channel Attacks, Secure Microarchitecture, Instruction Reordering

1 Introduction

Power, electromagnetic (EM), and temperature analysis attacks, also called side-channel attacks, exploit the physical parameters of the processor to extract secret information. The system continues to operate normally while being monitored, and no evidence of the attack is left behind [52]. Prior work focused on protecting small embedded micro-controllers and simple in-order processors dedicated to cryptographic applications [2, 3, 7, 11].

*Both authors contributed equally to this research.
[1]New Paper, Not an Extension of a Conference Paper.

Authors' Contact Information: Yun Chen, National University of Singapore, Singapore, yun.chen@u.nus.edu; Ali Hajiabadi, National University of Singapore, Singapore, ali.hajiabadi@u.nus.edu; Romain Poussier, Nanyang Technological University, Singapore, romain.poussiercr@gmail.com; Yaswanth Tavva, National University of Singapore, Singapore, yaswanth@u.nus.edu; Andreas Diavastos, National University of Singapore, Singapore, diavastos@gmail.com; Shivam Bhasin, Temasek Laboratories @ Nanyang Technological University, Singapore, sbhasin@ntu.edu.sg; Trevor E. Carlson, National University of Singapore, Singapore, tcarlson@comp.nus.edu.sg.

ACM 1544-3973/2024/1-ART1
https://doi.org/10.1145/3701991

However, recently we have seen power analysis attacks targeting more complex, high-performance processors based on out-of-order execution, found in mobile devices, desktops, and servers [18, 23, 47, 49, 65].

Power analysis attacks in particular exploit the synchronization of instruction execution and its correlation with power consumption (collected as power traces) to uncover secret information from a running application [24]. Such attacks are effective due to the deterministic behavior of the processing units, and as demonstrated by prior work [56], even modern general-purpose out-of-order processors show a high level of regularity, making them relatively easy to attack [49]. The number of power traces required to reveal secret information defines the level of security of the processor [8, 40, 76].

Many countermeasures have been proposed to combat power analysis attacks, such as just-in-time compilation, random code injection, and instruction descheduling, that obfuscate the execution with added power noise by randomizing the execution of instructions [2–4, 7, 11, 22, 41, 89]. Unfortunately, such countermeasures introduce large overheads in performance, power, and area, due to the unsupervised randomness and excess instructions executed. The overheads in performance and power reach up to 270%, while the area can be twice the size of the original processing unit. More importantly, they tend to be either application-specific requiring compiler support or target simple dedicated-to-encryption hardware that cannot be applied to high-performance general-purpose processors. To protect current general-purpose out-of-order processors, several software and firmware solutions have been proposed recently [49, 65]. However, they do not obfuscate the power side-channels of the processor at the source, and an attacker with physical access to the system can still collect high-resolution power traces. In addition, most of them assume a trusted Operating System (OS), but a successful attack on an Intel SGX enclave with a malicious OS has been shown in [49].

In this paper, we introduce Paradise, a new hardware-only 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. The key insight of Paradise is that to efficiently offer always-on protection in a general-purpose processor we must *randomize the ordering of instructions in such a way that minimizes the effect on the critical path of the execution* [27]. We propose a novel instruction scheduler that uses the time difference between the production of the two operands of an instruction (slack) to randomize the execution of non-critical instructions in a targeted way. Doing so desynchronizes the execution, increases noise in the power measurements, and obfuscates the power side-channel. The result is a low-overhead, general-purpose technique that uses always-on protection by obfuscating instruction execution to break the power consumption correlation that is normally observed by the adversary.

To better understand the lower bound of the security guarantees of the design, we introduce a comprehensive and reproducible framework for rapid security evaluation of early design choices. With our security evaluation setup, we demonstrate the dangers of using simplistic evaluation methods deployed in prior work that can give a false sense of security.

The main contributions of this work are as followed:

- A general-purpose out-of-order core design with always-on protection, called Paradise, that provides a significant security improvement against power analysis attacks demonstrated on AES-128 (556× when combined with Boolean Masking over a baseline only protected by masking, and 62, 500× over an unprotected out-of-order baseline for an advanced strong adversary);
- A dynamic instruction scheduling technique that combines instruction slack with randomness to provide low performance overhead (less than 3.2% on average as opposed to up to 270% performance overhead of other state-of-the-art solutions [2]), with negligible power and area overheads (1.2% and 0.8% respectively);
- A comprehensive security evaluation framework with a strong adversary with full knowledge of the countermeasure that complies with the highest security standards.

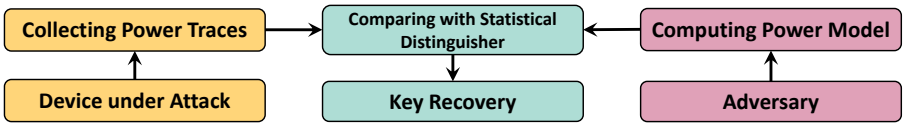

Fig. 1. Differential Power Analysis (DPA) flow. Power traces are collected and the adversary tries to compute the power model. A statistical distinguisher (correlation, difference of means, etc.) is used to recover the secret key.

2 Background and Motivation

2.1 Power Analysis Attacks

Power analysis attacks 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. In this section, we use the AES-128 algorithm as an example of how to recover secret keys from power traces.

In the first round of the AES-128, a 16-byte plaintext is loaded byte-by-byte and equation $I_n = Sbox[X_n \oplus K_n]$ is applied to form an intermediate state of the ciphertext (the 16-byte encrypted output of the algorithm). In this equation, I_n is the n^{th} byte of the intermediate value, 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. The Sbox and the plaintext (X_n) are both known to the adversary.

A typical power analysis attack, shown in Figure 1, targets a small part of the key or subkey[2]. First, the adversary needs to select a power model reflecting the power consumption behavior of the device. This can be done using a priori assumptions of leakage behavior like the Hamming weight/distance model [14] (better known as an unprofiled attack), or by trying to characterize the actual model (i.e. profiled attack), using, for example, Gaussian template [17] or supervised machine learning [63]. 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 on 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} ($16\ bytes \times 2^8$) when using a divide and conquer methodology[3]. Many different statistical distinguishers, or attack tools, have been introduced in the literature, with Kocher's Differential Power Analysis (DPA [46]), Correlation Power Analysis (CPA [14]), Mutual Information Analysis (MIA [10]) and the maximum likelihood template [17] being some examples. In the rest of the paper, 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.

2.2 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. Such attacks are effective due to the deterministic behavior of processing units. Thus, the security of a system is strongly dependent on how regular its power trace patterns are. More regular patterns make an attack easier to recover secret keys with a small and reasonable number of observations. As demonstrated by previous works [55, 56], even modern general-purpose out-of-order processors show a high level of regularity, making them easy to attack using a relatively small number of traces [49].

[2]Typically, for AES, the subkey is one byte, each attacked independently.
[3]By adjusting the variable I_n and device-specific power leakage model, the proposed framework can be adapted to other cryptographic algorithms.

Many countermeasures have been proposed to combat power analysis attacks [3, 4, 22, 37, 68, 89]. We categorize these countermeasures into two generic techniques: (1) *hiding* and (2) *masking*. *Hiding* limits or hides 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 relationships between power consumed are weakened for executed data or instructions or by randomizing the execution sequence.

Masking splits any sensitive intermediate variable into several statistically independent shares, similar to the principles of Shamir's secret sharing [70]. An adversary can learn nothing about the sensitive intermediate variable unless all the shares are available. While masking can provide strong guarantees from a cryptographic point of view, it is effective only in the presence of noise [33]. Thus, masking and hiding are complementary countermeasures, where hiding provides the ideal low SNR environment for masking to be effective.

The goal of this work is to efficiently enhance secret *hiding* on the execution unit and the register file of a general-purpose out-of-order processor, which have the greatest impact on the execution's power signature, making them an easier target to attack [2]. Our target is to leverage the dynamic scheduling of the out-of-order processors to randomly delay the issue and execution of non-critical instructions, hence, providing different instruction schedules in every run. This results in a non-deterministic behavior of the core that ultimately hides secret information.

3 Threat Model and Assumptions

Adversaries' assumptions. In this work, we assume several adversaries with different levels of capabilities (*Basic* with no knowledge of the countermeasure, *Educated* with limited knowledge, and *Advanced* with full knowledge of the Paradise system). For the strongest adversary, we also assume they have access to the device under attack for which they can control everything including the randomness algorithm. This attacker can learn the precise leakage model, leading to a confident lower-bound on security to defeat Paradise. The attacker can collect power traces either through physical access to the device, for example using probes, or through remote access and software-based interfaces as demonstrated in [49, 54] for AMD and Intel x86 CPUs. Although we only consider power attacks, Paradise can harden other physical side-channels with similar attack methodologies, like electromagnetic (EM).

Applications and attack surface. Paradise aims to resist power side-channel attacks (SCA) that extract private cryptographic keys, which are the most dangerous and critical threats for physical side-channels and also tougher to mitigate. We assume a safe implementation of the cryptographic algorithm is being used (*e.g.*, a constant-time implementation), and that there are no secret-dependent timing and access patterns in a single execution[4]. We evaluate Paradise with an AES-128 algorithm as it is commonly used in SCA literature. Public key cryptography algorithms, such as RSA, are also vulnerable to power SCA and previous work [86] has shown that jitter or misalignment (as used by Paradise) improves SCA resistance.

4 Paradise Microarchitecture

Paradise is an out-of-order processor that implements a novel, efficient instruction scheduler that detects and uses the time between an instructions operand availability (slack) to randomly delay non-critical instructions at runtime. These instructions are delayed before being issued to randomize their execution and their access to the register file or memory. This desynchronizes the execution with little effect on the critical path, hence maintaining high performance.

[4]Unsafe implementations with secret-dependent timing and access patterns are vulnerable to a wide range of attacks, like timing and cache side-channels, that the attacker would deploy them before considering power analysis attacks.

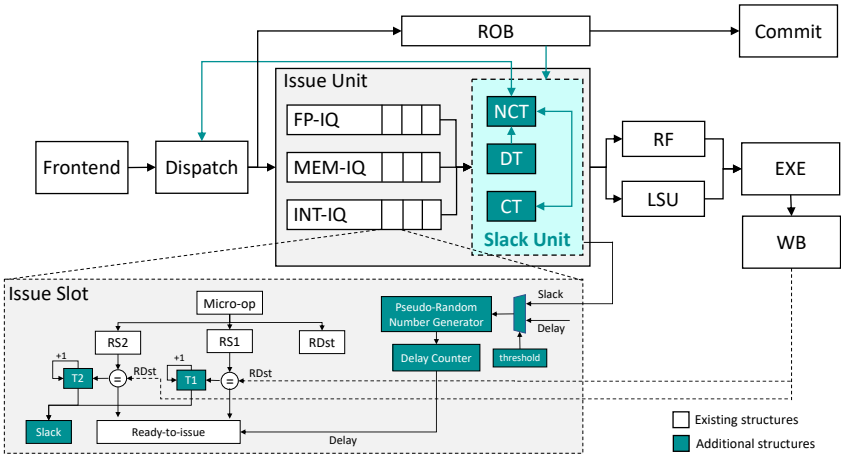

Fig. 2. The microarchitecture of PARADISE.

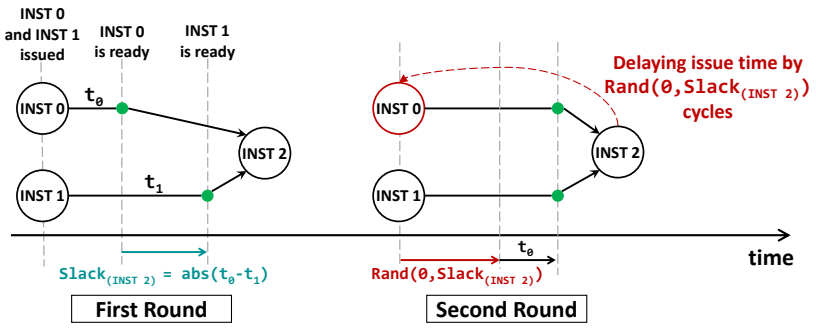

Fig. 3. The slack of INST 2 is the absolute difference between the times that INST 0 and INST 1 produce their result (left). This slack is then used to delay the non-critical producer (INST 0), in its next appearance, to change its issue time without affecting the issue time of INST 2 (right).

PARADISE is built on top of SonicBOOM [91][5], a RISC-V out-of-order core with a 7-stage pipeline; it implements a new structure called the Slack Unit and updates the Issue Slots to dynamically record the slack of instructions and communicate the delay to be applied to selected, typically non-critical, instructions (Figure 2).

4.1 Definitions of Instruction Criticality and Slack

Similarly to previous work [27], we define criticality by the time the input operands of an instruction are produced. A non-critical instruction is one that if delayed, will not affect the execution start time of its consumers, as long as the delay does not exceed the extra time it takes for the second operand to arrive. We define this time difference between the production of an instruction's operands as *slack*. In the example of Figure 3, 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 Equation 1.

$$slack(INST\ 2) = abs(t_0 - t_1) \tag{1}$$

[5]We have open-sourced PARADISE implementation in https://github.com/PARADISE-Secure-Core/PARADISE-Core.

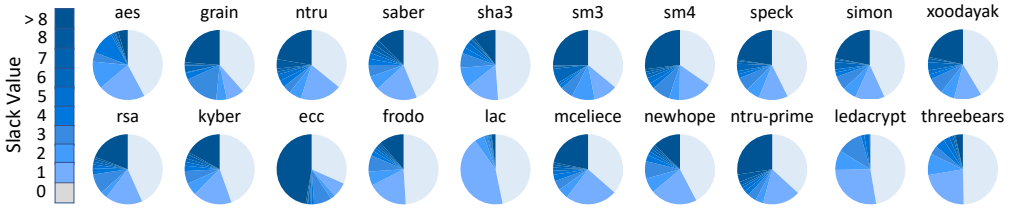

Fig. 4. Slack distribution of dynamic instructions for security-related applications. On average 58% of dynamic instructions have slack (stable slack for 83% of them).

Destination Table (DT)

PC (12 bits)	Non-critical PC offset (8 bits)
0x1310	192
0x1330	8
...	...

Critical Table (CT)

PC (12 bits)
0x1400
0x1420
...

Non-Critical Table (NCT)

PC (12 bits)	Slack (5 bits)	Stable (1 bit)
0x1210	10	0
0x1220	5	1
...	...	...

Fig. 5. The Destination Table (DT) and the Critical Table (CT) store currently dispatched and critical instructions, respectively. The Non-Critical Table (NCT) stores non-critical instructions and the smallest slack observed;

In the example of Figure 3, INST 0 and INST 1 produce their data for INST 2 at times t_0 and t_1 respectively. 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.

We investigated a set of security-related applications (NIST post-quantum candidates and lightweight cryptography algorithms [32, 57, 74]) to determine the amount of available slack. Figure 4 shows that there is enough slack between dynamic instructions to efficiently enable the proposed countermeasure. At least 50% of every application's dynamic instructions have a non-zero slack (on average 58% across all security applications); delay is injected for 47% of them with stable slack for 83% of the delayed instructions.

In the rare cases that not enough slack exists, PARADISE injects a predefined maximum slack to provide always-on protection and guarantee security.

4.2 Slack Unit

The Slack Unit is introduced in the design to store runtime information in order to learn the slack and inject appropriate delays when needed to randomize instruction issue. It is built using three set-associative memory structures[6] (see Figure 5):

(1) Destination Table (DT) holds issued instructions and their non-critical producer. The DT is queried by the issued instructions to check if their non-critical producers are the same in subsequent appearances.

(2) Critical Table (CT) holds instructions that were marked as critical, to handle criticality conflicts when an instruction is a critical producer for one instruction but a non-critical producer for another.

(3) Non-Critical Table (NCT): Holds non-critical instructions and their corresponding slack. For performance reasons, we also mark whether the instruction has consistently been 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.

Testing multiple applications indicates that an 8-bit offset is sufficient to represent the PCs of non-critical instructions in the DT. To avoid introducing new security vulnerabilities when context-switching processes access the Slack Unit with the same PCs (false hits), we flush it on context switching.

[6]Using a Bit Pseudo Least Recently Used replacement policy [50].

4.3 Detection of Criticality and Slack

When a new instruction is issued, an entry is allocated for it in the Destination Table (DT). When it completes its execution, the Slack Unit detects the criticality of its producers and calculates the slack that will be used to generate the random delay for the corresponding non-critical instruction in its next appearance. 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 more than one instruction 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, we only keep the smallest slack value to minimize the effect on the critical path. 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 changes, we make the appropriate changes in all Slack Unit structures. It usually requires a few iterations to learn the slack (see Section 4.6 for an illustrative example and Figures 13 and 14 for the average number of required iterations to learn the slack in SPEC CPU2017 and security-critical applications). For example, `deepsjeng` application in SPEC CPU2017 requires only one iteration to learn the slack for 51% of the instructions, while only 4% of the instructions require more than 20 iterations.

4.4 Measuring Slack and Randomizing Execution

To learn the slack of executing instructions, we need to track the instruction dependencies and their execution times. To track dependencies, we compare the source registers of the instructions (RS1 and RS2 in Figure 2) in all issue slots with the destination register of issued instructions. To measure the production times of their source registers, we update each issue slot with two timer fields (T1 and T2) that count the number of cycles each source register took to resolve.

To randomize the execution of non-critical instructions, we inject a random delay[7] to their issue time within the range of the appropriate slack. A Delay Counter starts a countdown with this delay only after all instruction dependencies are resolved. When the countdown is complete, it marks the instruction as Ready-to-Issue. When issued for execution, its producers and their new slack are stored in the Slack Unit.

4.5 Delay Injection

To find whether a newly dispatched instruction is to be delayed, the Slack Unit is queried (specifically the NCT) with the instruction PC and in case of a hit, the appropriate delay is returned and injected to its issue slot. As soon as its operands are produced, the instruction will be marked to wait for another *delay* number of cycles. If the corresponding slack in NCT is marked as `stable`, it will be used as an upper bound to select a random value between 0 and the stable slack value. If the slack is not stable, we label it as being in the *unstable phase*, and we don't randomize the delay. Instead, we will use the slack value as is for the delay. Note that, the Slack Unit is constantly learning to determine a stable slack.

PARADISE is an always-on protection design and tries to obfuscate all instructions regardless of the slack available. To provide this security guarantee the Slack Unit monitors the accumulated slack in an instruction window (a set of live instructions) to determine whether additional slack, above that available in the application itself, is needed. If the available slack is below 100 cycles for all instructions in the Slack Unit, we temporarily enable a security model called *random-iso-security* that randomly injects up to 8 cycles delay with 20% probability

[7]We use the Galois Linear Feedback Shift Register (GaloisLFSR) function [53]. Alternative random number generators exist [85], but we choose the best tool at our disposal to implement in RISC-V SonicBOOM.

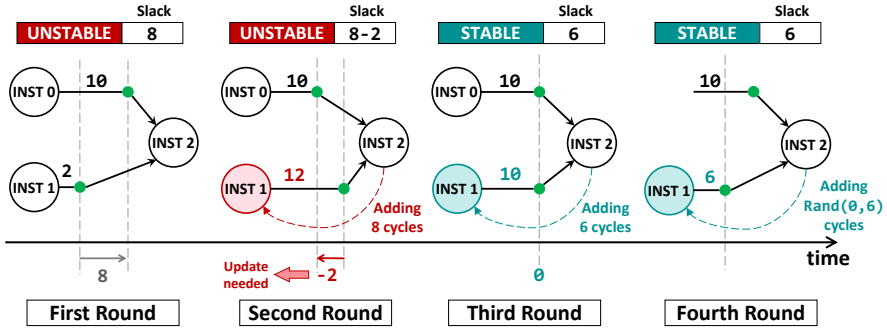

Fig. 6. Slack detection and delay injection example, showing the stability of slack in multiple rounds. The instability of INST 1 in the first appearances is common due to backward dependencies to previous instructions that might have been delayed as well.

per instruction. This model achieves the same security-level as PARADISE on AES and is described and evaluated in detail in Sections 6 and 7. The performance penalty, in this case, is negligible as the time that the random-iso-security model is enabled is small (~ 0.1% of the execution in our results which corresponds to the slack learning phase).

These modifications seamlessly run in parallel with the rest of the processor operations and do not affect the pipeline stages.

4.6 Example

Figure 6 shows an example of how we detect instruction criticality and slack and how we inject a delay on non-critical instructions. When INST 2 is issued in the first round, we detect the completion time for each of its input operands and calculate the slack (8 cycles). INST 1 is the critical instruction that we can't delay as it finished last and INST 0 is marked as non-critical with a slack of 8 cycles. Because INST 0 is still unstable, in its next appearance (second round) we inject the computed slack as is. Although we expected the new slack of INST 2 to be 0 as INST 0 still needs 10 cycles to complete, we find that INST 0 took 12 cycles to finish in this round (instead of the 8 cycles that we injected as a delay). This is relatively common for all instructions that appear for the first time due to the backward dependencies of INST 0 to previous instructions that might have also been delayed. We call this the *unstable phase* where the instruction is marked as unstable and the Slack Unit records the slack for INST 0 as the difference between the old and the new slack. When the unstable phase is over in the third round, the instruction will be marked as stable and the injected delay thereafter (fourth round) will be randomized with the updated slack as the upper bound.

5 Security Evaluation Methodology

In this section, we propose a comprehensive security evaluation framework to better identify the lower bound of security for our newly proposed hardware design. We first discuss the dangers of the commonly used, simplistic pass-fail security evaluation methods, and then describe our proposed security framework and metrics.

5.1 Limitations of Simple Pass-Fail Tests

T-test: The amount of leakage is defined by the statistical moment on 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 information lies in the mean

of the side-channel traces, forcing an adversary to extract information in (at least) the variance and covariance. The Test Vector Leakage Assessment (TVLA [30]) methodology was introduced to detect the presence of this first-order leakage for masked implementations and can be extended for higher orders [19]. That is, a T-test failure shows the existence of such leakage, while 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 [8, 76]. 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 *hiding* countermeasures, where leakages would exist at first-order, should be avoided.

Fail/success unprofiled CPA: The strengths and weaknesses of an unprofiled CPA lie in its assumptions. On one hand, it is designed to process one-time samples at a time (also known as univariate leakage [76]). While one can combine time samples together before performing CPA (i.e. multivariate attack [76]), doing so is sub-optimal. On the other hand, CPA assumes some leakage model and is often a good security estimate for unprotected implementations where the side-channel traces are well aligned and the power model is well known (e.g. Hamming weight/distance). However, this no longer holds when the type of leakage diverges from these assumptions. An implementation that changes the underlying leakage model, 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 applies measurement alignment/processing techniques) might achieve key recovery with significantly fewer observations. In Section 7.1, we demonstrate the dangers of using a basic CPA attack in evaluating PARADISE, and in Section 5.2, we propose an evaluation framework that considers more powerful adversaries that are capable of applying the right leakage model and processing techniques to exploit the available leakage in an optimal manner. In addition, we discuss the same limitations in the evaluation of prior work in Section 9.2.

5.2 Proposed Evaluation Framework

The limitations of the previously discussed methods can be summarized as using weak or incorrect adversarial assumptions. 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 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 [17]. As a result, evaluating the security of a device with the wrong model and method can lead to a false sense of security. 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.

We will illustrate this by providing different levels of security analysis for three types of adversaries, which we call (1) *basic*, (2) *educated*, and (3) *advanced*. First, in order to compare our work to existing literature, the **basic** evaluation considers an adversary that applies standard CPA with a Hamming weight model on the Sbox output [14]. Second, as part of the security introduced 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 change can greatly alter 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. We will first take advantage of a profiling (or training) set in order to mount a multivariate template attack [17] 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 covariance matrix. A 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.

5.3 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 are denoted by $\mathbf{p}$. In our evaluation, we will compute the security using the following metrics:

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

Guessing entropy [77]: 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 $\mathsf{rank}(\mathbf{p}, k)$ 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 2 allows one to compute the guessing entropy.

$$\mathrm{GE} = \frac{\sum_{i=0}^{n_a-1} \mathsf{rank}(\mathbf{p}_i, k)}{n_a}. \tag{2}$$

The use of guessing entropy provides more information than the commonly used measurement to disclosure (MtD) metric [51]. 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 shows the global key recovery progression instead of only reporting the overall number of traces. 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 [66]. Looking only at the number of measurements required to recover the key can be misleading, as the implementation might be broken with fewer traces using brute force or key enumeration.

6 Experimental Setup

We implement Paradise on top of SonicBOOM [91], an open-source RISC-V out-of-order processor[8]. We used the largest possible core configuration (AWS FPGA-limited) as shown in Table 1.

Apart from the out-of-order baseline and Paradise, we also implement three generic processors to validate the importance of using targeted injected delays (Table 2). For each of these processors, random delays (up to 8 cycles) are injected to each instruction with a given probability to either match the performance of Paradise (*random-iso-perf*), or the level of security (*random-iso-security*).

Benchmarks. To evaluate the performance impacts, we run the microbenchmarks provided by Chipyard [5]. We then use FireSim [43] to run the 10 supported SPEC CPU2017 benchmarks. We demonstrate the security of designs in Table 2 by targeting AES-128 [1].

Area and Power. We use the Synopsys Design Compiler (DC) [78] to synthesize Paradise and SonicBOOM using a commercial 22nm process. Using VCS [80], we conduct gate-level simulations on synthesized processors

[8]SonicBOOM is the most recent and performant version of BOOM microarchitecture at the time of this submission [81].

Table 1. System Configuration

RF/Fetch Buffer/ROB	2x128/24/96 entries	L2 Cache	4 MB, 8-way set assoc.
Issue Queue	3×16 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, TAGE	*(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.		

Table 2. Processor Implementation Details

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

for realistic activity. PrimePower [79] analyzes gate-level waveform and code to determine power consumption of PARADISE and its overhead compared to the *ooo-baseline*. Note, that we do not use PrimePower to collect power traces for security analysis. The rest of this section discusses our power simulation framework for power trace acquisition and provides a detailed discussion on the shortcomings of existing alternative methods.

Power Traces. 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 plaintexts. The second set (profiling set), is composed of randomly varying keys and plaintexts that are known by the adversary to perform advanced profiling.

Power Trace Acquisition. For our proposed comprehensive security evaluation, we require detailed and fine-grained power traces. We deploy a power simulation framework to estimate the power consumption that is *consistently correlated* with physically measured power. We collect information at the instruction level from the behavioral simulation of the CPU, to perform rapid and correct power simulation (similar to [71, 72]). We collect millions of power traces and then use a Hamming weight leakage model to estimate the power. State-of-the-art research shows that this model is robustly correlated with the physical power consumption [2, 49, 58, 59, 67]. Equation 3 shows the power estimation of our simulator at any given time T.

$$\text{power}(T) = \begin{cases} 0 & \nexists \quad \text{inst} : \text{inst.WB} = T \\ \sum_{\text{inst}} \text{HW}(\text{inst}) & \forall \quad \text{inst} : \text{inst.WB} = T \end{cases} \tag{3}$$

The write-back time of instruction inst is denoted by inst.WB. Also, HW(inst) is the Hamming weight of the data that instruction inst writes to the system. Note that these are best-case settings providing maximum available leakage to an attacker. In a physical chip, the leakage would likely have a signal-to-noise ratio that is orders of magnitude worse, scaling up the security margins accordingly.

Alternatives. There are several alternative techniques for collecting power traces for a power SCA evaluation: (1) Using a real system and connecting an oscilloscope to the circuit [87]. However, this requires several rounds of fabrication of the designed processor followed by evaluating its security which is costly and time-consuming, and more importantly, not practical for rapid evaluation of multiple early design choices. (2) PrimePower can simulate the cycle-accurate power consumption at the gate level. However, the simulation speed of PrimePower is extremely slow (20 traces per day for SonicBOOM) and it would take years to produce the millions of traces required for a comprehensive security evaluation. Using multiple instances of PrimePower is possible but the cost of multiple licenses is prohibitive [36, 61]. (3) Mapping and executing the processor on an FPGA allows the collection of power traces from the FPGA power monitor port using an oscilloscope. However, we discovered several limitations in our efforts when using a state-of-the-art FPGA (the Sakura-X kintex-7 FPGA board [35]): (1) A realistic design of SonicBOOM does not fit on the FPGA: the largest configuration that can fit has a decode/issue/commit width of only 1 instruction, which is not a realistic out-of-order processor. (2) A state-of-the-art Keysight oscilloscope [44] does not allow a full run of AES-128 with a million random plaintexts, as required by the proposed security evaluation framework. It can only store 2 million sample points per file when operating at 20MHz; (3) Using FPGA traces we can only run a basic CPA attack. An advanced security evaluation like the one we propose, however, requires fine-grained information about the AES-128 code running on the core (*e.g.*, the exact timing of the SBox operations). Finally, (4) synthesizing an out-of-order core for the FPGA will only match the original hardware on a logical level but not on the netlist level. The resulting lookup tables used in the FPGA lose certain characteristics of the out-of-order core and separation of different components, ultimately losing the actual leakages of the core; the state-of-the-art work could only manually map the register file of an in-order core with two-stage pipeline [29], which is impractical for SonicBOOM. Currently, there is no verified tool that translates out-of-order core designs to FPGA with validated and correlated leakage assessment.

We choose to use a verified simulation framework that enables researchers to perform a comprehensive, rapid, and reproducible evaluation of multiple and early designs with realistic configurations.

7 Experimental Results

7.1 Security Evaluation

For the security evaluation, we consider the three types of adversaries that we introduced in Section 5.2 to extract the secret key of AES-128. For each method, we compare the security benefits of each core with their corresponding *ooo-baseline* with respect to the same attack. Our goal with each enhanced evaluation technique is to demonstrate how more knowledge about the type of countermeasures in place can easily beat the current security techniques, ultimately showing that basic evaluation methods give a false sense of security.

Basic evaluation. We collected 10 million attack traces and divided them into 20 subsets, performed a standard CPA on each of them, and averaged them to compute the guessing entropy (Figure 7(a)). The x-axis corresponds to the number of traces and the y-axis to the guessing entropy. A guessing entropy of 0 indicates that the correct key is the highest ranked (on average) and thus the attack is successful.

For the *io-baseline* and the *ooo-baseline* we recover the key with 500 and 1,800 traces respectively. Indeed, the out-of-order core provides some randomness in the computation timings but these results show only a small security benefit. We use the *ooo-baseline* hereafter, as the unprotected baseline for our security evaluation. We observe that the Paradise countermeasure shows a security of 261× over the *ooo-baseline*, requiring 470,000 traces when using standard CPA. However, *random-iso-perf* design (i.e., adding random delays with the same performance as Paradise) provides security of 12× (requiring only 22,000 traces for key recovery), which shows that the Paradise method shows greater security benefits.

Finally, *random-aggressive* only requires 220,000 traces, which corresponds to a security benefit of 122×. While the benefits are in the same order of magnitude as Paradise, the difference can be explained when looking at the

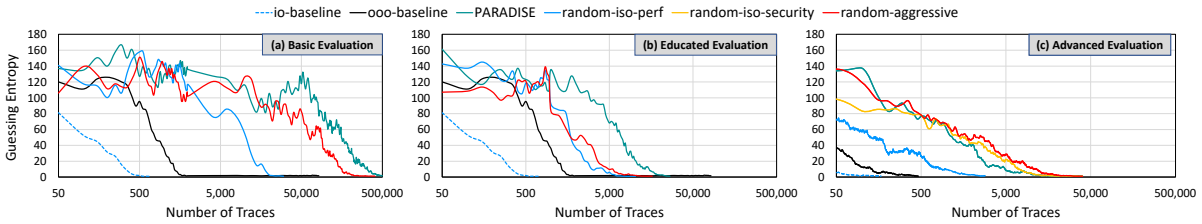

Fig. 7. Results of (a) the basic security evaluation, (b) the educated security evaluation, (c) the advanced security evaluation. The x-axis corresponds to the number of traces (in $\log_{10}$ scale), and the y-axis corresponds to guessing entropy.

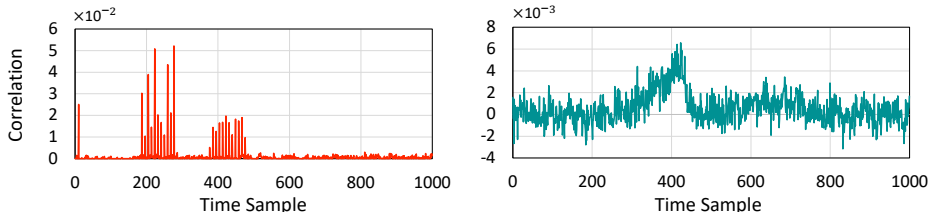

Fig. 8. Leaking regions using profiled CPA for *ooo-baseline* (left) and *random-aggressive* (right). The x-axis corresponds to the time samples, and the y-axis corresponds to the correlation. Note the y-axis scale differences.

two security benefits introduced by our countermeasure. The security of PARADISE mainly comes from (1) targeted desynchronization and (2) more randomness in the register's content. The *random-aggressive* implementation focuses on increasing the desynchronization, but due to its policy to add random delays to all instructions, we observe that the relative instruction order appears to remain more intact, which results in fewer differences in the register file content. In addition, because *random-aggressive* has no information on the slack, it must use a maximum delay upper bound (8 cycles in this case), while PARADISE is only limited to the slack observed (up to 25 cycles for AES).

Educated evaluation. As a second evaluation, we illustrate how some basic knowledge of the type of countermeasure and a simple optimization of the attack itself can drastically change the outcome. A significant part of the security introduced by our countermeasure comes from desynchronization, thus 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(b).

We do not report any results for *io-baseline* and *ooo-baseline* implementations, as basic evaluation was better (in terms of number of traces required) for non-existent and limited desynchronization. For that reason, we will 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 are now broken with 22,250 (12×) and 20,000 (11×) traces respectively. This simple optimization demonstrates the danger of using sub-optimal attack strategies for security evaluations.

Advanced evaluation. Our third 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 [26] in order to identify leaking features in the trace. The results are shown in Figure 8, where

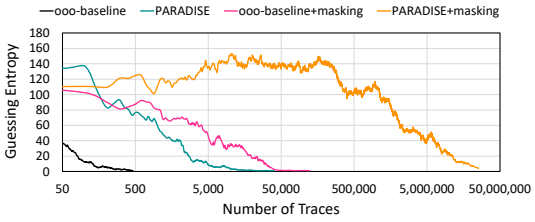

Fig. 9. The advanced security evaluation including masking. The x-axis corresponds to the number of traces (in $\log_{10}$ scale) and the y-axis to the guessing entropy.

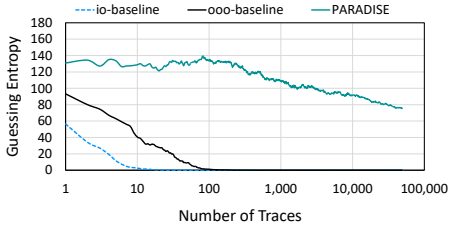

Fig. 10. CNN-based attack. The x-axis corresponds to the number of traces (in $\log_{10}$ scale) and the y-axis to the guessing entropy.

the left graph shows the results for the *ooo-baseline* and the right graph shows results for *random-aggressive* implementations.

Leakages are clearly identified with several peaks for the *ooo-baseline* implementation. We observe similar behavior for the *io-baseline* and *random-iso-perf*. For these three 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 also observed for Paradise and *random-iso-security* implementations, due to the desynchronization brought by the countermeasures. In that case, we selected all-time samples happening before and after 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 [17]. For each implementation, we use different numbers of principal components, and show the best results for each of them in Figure 7(c). We also show the results for the *random-iso-security* implementation, having similar security to Paradise (tailored specifically in the case of the advanced evaluation).

The advanced method produces better results in terms of attack power than the basic and educated ones. First, the *io-baseline* and *ooo-baseline* implementations now only require 125 and 400 traces respectively. Paradise now requires 13,500 traces, showing a security gain of 34× (compared to the *ooo-baseline* with the same adversary). However, *random-iso-perf* can now be broken with 2,200 traces, hinting that our countermeasure is 5.5× 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 Paradise, with a benefit of 38×. Indeed, as we now profile the leakage model, the effect of the vertical noise is reduced, which has more impact on Paradise. Overall, this shows that using the wrong or a 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 Paradise, and by 15 for the *random-aggressive* core due to wrong model assumptions. However, the number of traces needed for the unprotected *ooo-baseline* implementation was only divided by 4.5, as the model was already fitting more. Note, that we tested multiple and different numbers of principal components for each implementation (i.e., different traces' division), and we report the numbers that achieve the best attack results.

Masking Evaluation. To highlight the benefits of combing Paradise with masking, we simulated side-channel leakage corresponding to a lookup table implementation of first-order Boolean Masking [20]. The attacker is provided with two leakages corresponding to the Sbox output sharing $\mathrm{S}(k \oplus x) \oplus m$ and m, where m is a secret mask, uniformly chosen at random and changing at each execution. To reflect the type of implementation used, the SNR of both leakages is chosen according to the SNR obtained during the advanced evaluation for both

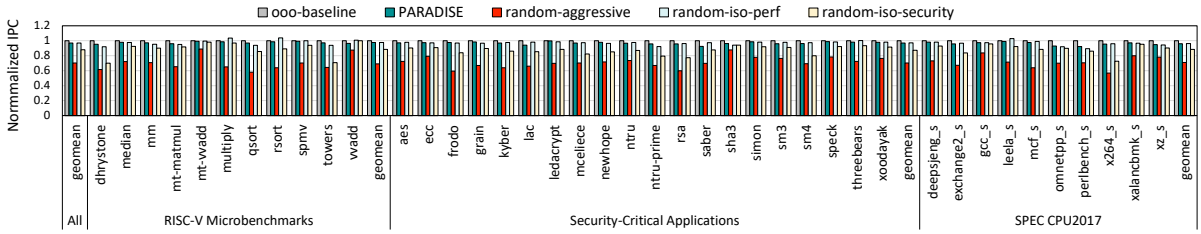

Fig. 11. Performance normalized to the *ooo-baseline*. We run all SPEC CPU2017 benchmarks compatible with FireSim.

ooo-baseline (0.014) and PARADISE (0.00057). Using these leakage values, we performed the *advanced evaluation*. The corresponding results are shown in Figure 9. Indeed, 45,000 traces are now required for *ooo-baseline+masking* (112× compared to *ooo-baseline*), while 25,000,000 are needed to break PARADISE+*masking* (556× compared to *ooo-baseline+masking*, and 62, 500× compared to *ooo-baseline*). This difference can be explained by the quadratic benefit in terms of the number of traces needed when the noise increases from using 2-share masking [33].

Note, that the number of traces presented in this section is very small compared to non-simulated scenarios as our simulated traces have zero noise. Prior work [49] shows that 16 million traces (taking 11.5 days) were needed to reconstruct 12 of 16 bytes of the AES-128 secret key, from an Intel server, out-of-order processor. We believe that the combination of PARADISE and masking would eradicate the dangers of power analysis attacks in real hardware with the extra noise present in the measurements.

CNN-based SCA. Recent work in profiled SCA attacks have seen rapid adoption of deep learning techniques [13, 63], in particular convolution neural networks (CNNs), as the transitional invariance property of CNNs is widely exploited by side-channel attacks to overcome countermeasures like jitter [16]. As the transitional invariance property of convolution neural networks (CNNs) is exploited by SCAs to overcome countermeasures [16], we evaluated the performance of CNN-based attack [63] against the most representative implementations: *io-baseline*, *ooo-baseline*, and PARADISE. We used an approach similar to [90] on the AES_RD dataset with three convolution layers, targeting random delay countermeasures (similar to PARADISE). Three convolution layers are as follows with respect to channel count (C), kernel size (K), padding on each side (D), average pool size (P), and pool stride (S): (layer 1: C=4, K=1, P=2, S=2; layer 2: C=8, K=50, P=50, D=24, P=50, S=50; layer 3: C=16, K=3, D=1, P=7, S=7). We use the Glorot weight initialization [31], the One Cycle Policy [73] with a learning rate of 0.0025 to 0.005, and the Adam optimizer [45], adapted to our dataset. We train for 100 epochs with a 90:10 training/validation split and report the guessing entropy in Figure 10. The *ooo-baseline* was broken in 588 traces with CNN as compared to 400 with the advanced evaluation. However, for a protected dataset like PARADISE, the CNN does not converge.

Finding the best CNN architecture for a particular dataset is an open problem in side-channel literature. Thus, the results obtained with CNNs cannot be claimed to be a worst-case analysis as one could find a better CNN architecture. Moreover, template attacks, like our advanced evaluation, have been theoretically proven to be the optimal attack from an information theory point of view [9].

7.2 Performance Evaluation

We evaluated several configurations for the Slack Unit in PARADISE by running an AES-128 encryption engine to find the best combination of performance, power, and area, and we found that a 4-way, 64-entry cache implementation provides the best combination. Figure 12 confirms 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 12 are calculated as $\frac{overhead_{perf}}{overhead_{power}}$ and $\frac{overhead_{perf}}{overhead_{area}}$. We use the highest number for $overhead_{perf}$ because in this scenario PARADISE is going through a longer *unstable phase* and although it injects more unstable delays, it

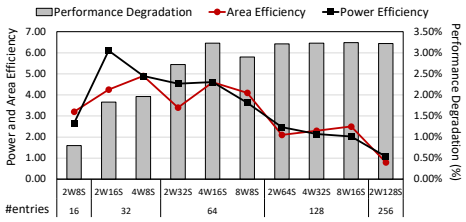

Fig. 12. Performance degradation compared to ooo-baseline core and area/power efficiency using different parameters. W and S represent number of *ways* and *sets* respectively, the *#entries* are equal to $W \times S$, which denote the total number of *entries* of each table in the Slack Unit.

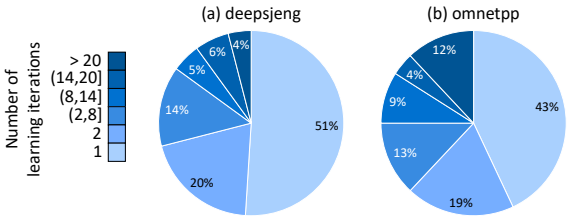

Fig. 13. Comparison of number of iterations required for Paradise to learn stable slack values in two different SPEC CPU2017 applications with low impact on performance (i.e., deepsjeng with 1.9% overhead) and high impact on performance (i.e., omnetpp with 7.0% overhead).

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.

Figure 11 shows the performance of Paradise (normalized IPC to the ooo-baseline) on three different sets of benchmarks (RISC-V bare-metal microbenchmarks, security-critical applications and SPEC CPU2017). For all the applications evaluated, the average performance overhead of Paradise is 3.2%. Performance overhead of random-iso-perf, random-iso-security, and random-aggressive implementations are 3.0%, 12.0%, and 29.9% overheads, respectively. An in-depth analysis showed that the overheads mostly come from the *unstable phase* where we inject an unstable delay. As soon as the criticality of the instructions and the slack becomes stable, the performance of the processor returns to its baseline levels. To demonstrate this point, Figure 13 depicts the number of required iterations in two SPEC CPU2017 applications to learn stable slack: (a) `deepsjeng` that experiences low impact on performance in Paradise (i.e., 1.9% performance overhead), and (b) `omnetpp` that experiences high impact on performance (i.e., 7.0% performance overhead). As shown in Figure 13(a), only one iteration is needed for 51% of the instructions in `deepsjeng` to learn the stable slack, while this percentage is reduced to 43% in `omnetpp` (see Figure 13b). Moreover, 12% of the instructions in `omnetpp` require more than 20 iterations to learn stable stack, while only 4% of the instructions in `deepsjeng` require more than 20 iterations.

In addition, the results in Figure 14 show that the vast majority of the security-critical applications evaluated in this work require less than 8 iterations to find stable slack values. Note, that the Paradise methodology aims to minimize performance overhead for critical paths of the programs that execute frequently; these hot regions of the programs provide ample opportunities to learn their stable slack values. In other words, infrequent regions of the programs do not contribute to the overall performance significantly and it is not critical to learn their stable slack.

The random-iso-security injects on average 0.9 cycles delay per instruction with a total of 86.4 cycles for 96 instructions (ROB size). This suggests that at least 86.4 cycles of accumulated slack over an instruction window is required to achieve an acceptable level of security. To guarantee this in Paradise when not enough slack is available we increase the aggressiveness of the delay and enable the random-iso-security if the available slack in the Slack Unit is below 100 cycles (threshold). Because all security applications tested have enough slack, we see an average of just 0.1% of cycles (Figure 15) where the slack is below the threshold, i.e., random-iso-security is enabled merely 0.1% of the time, on average; this overhead corresponds to the time needed for the slack learning phase.

Small core efficiency. While we use the largest provided configuration of SonicBOOM for the main results (see Table 1 for the details), we have also experimented a smaller configuration as shown in Table 3. Our results show that the small core has a performance overhead of 4.3% for Paradise when compared to an unprotected

Table 3. Small Core Configuration

RF	2×54 entries	Issue Queue	3×8 entries
Fetch Buffer/ROB	16/64	Execution Units	3 (1 MEM, 1 ALUs, 1 FPU)

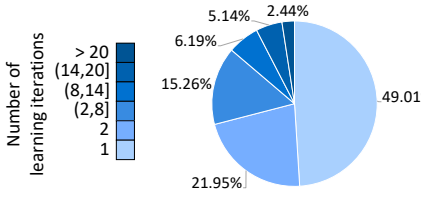

Fig. 14. Average number of iterations required for security-critical applications to learn stable slack.

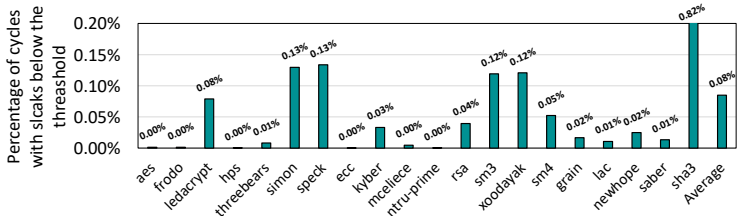

Fig. 15. Percentage of cycles that the random-iso-security is enabled.

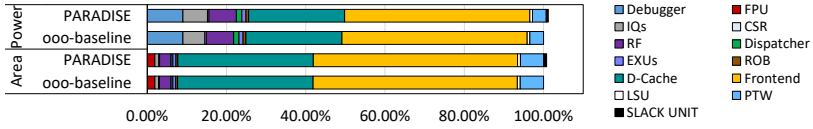

Fig. 16. Power and area overheads of PARADISE.

baseline. In addition, PARADISE has a power and area overhead of 2% and 1.03% with the small core configuration, respectively. These results show slightly higher overheads compared to the large core configuration, but they are still fairly close and acceptable.

7.3 Power and Area Overheads

Figure 16 shows the power and area of PARADISE compared to the *ooo-baseline*. The Slack Unit and the GaloisLFSRs (for randomizing the delay) introduce a negligible overhead of 1.2% for PARADISE. In matters of area, the total overhead of PARADISE compared to the ooo-baseline is 0.8%.

8 Discussion

8.1 PARADISE against Privacy Attacks

While PARADISE aims to resist power analysis attacks (i.e., extracting cryptographic keys), another use case of power side channels is leaking private information about the applications running on the processor (e.g., detecting the running application [38, 48, 88]). Defending against such attacks requires obfuscating the original power signature of the running application. To demonstrate that PARADISE also resists against such attacks, we run four security-related applications on both the unprotected *ooo-baseline* and PARADISE. Figure 17 shows the averaged power signals of 2,000 traces performing encryption using four different lightweight ciphers: Xoodyak [21], Simon [12], SM3 and SM4 [25] (these applications are also a subset of security-critical applications presented in Figure 4 and Figure 11) [9]. Results show that the *ooo-baseline* core is highly deterministic and different phases of the program are visible. On the other hand, PARADISE shows a high degree of obfuscation to hide the power signals of the running application. Additionally, Figure 18 depicts the box plots of the averaged power signals,

[9]Results of this section are inspired by Maya [65].

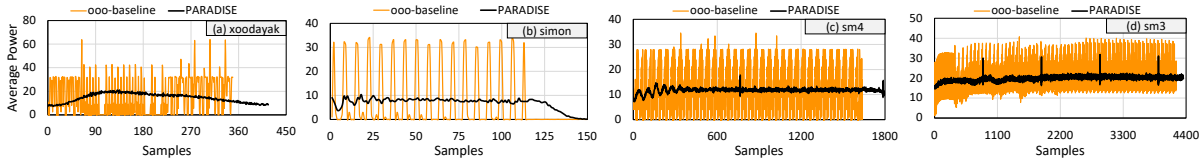

Fig. 17. Average power for 2,000 traces. The x-axis corresponds to the time samples, and the y-axis corresponds to the average power (see Equation 3).

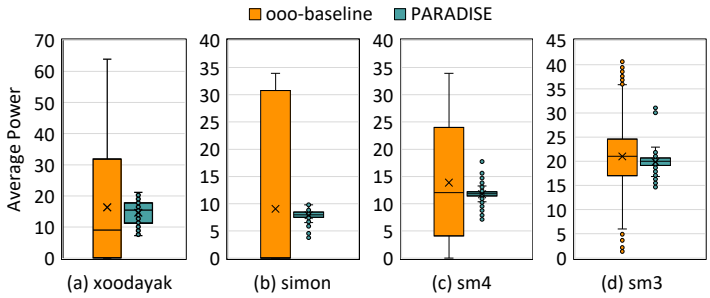

Fig. 18. Summary statistics of the average 2,000 power signals in Figure 17.

indicating the distribution of the values. While the power values are very diverse in the *ooo-baseline* and different applications have distinguishable distributions, PARADISE can hide this diversity of values. The target of PARADISE is not privacy attacks, but our findings confirm that the PARADISE methodology is capable of obfuscating power values for different applications, and potentially, making the privacy defenses [65] easier since most power signals and highly distinguishable patterns of the applications are hardened.

8.2 PARADISE-selective: Compiler-Informed Protection

We also analyze a version of PARADISE that enables protection for only secret-dependent regions, determined by static program analysis. We call this PARADISE-selective and we compare its security with PARADISE in Figure 19. PARADISE-selective requires 262,000 and 18,500 and 12,000 traces to recover the key in our basic, educated, and advanced security evaluations respectively. This translates to 145× improvement over *ooo-baseline* in basic evaluation, 10× in educated evaluation, and 30× in advanced evaluation. Results show that PARADISE with always-on protection provides better security guarantees because it is able to randomize a larger number of instructions and blend the execution of secret-dependent regions with the rest of the code to produce more noise overall. Another benefit of PARADISE over PARADISE-selective is that it does not require static analysis and re-compilation of all applications that need to be protected. Neither does it need the extra cost to communicate the compiler-driven information to the hardware.

9 Related Work

Table 4 summarizes the state-of-the-art works in terms of our evaluation metrics and other design features.

9.1 Power Analysis Attack Countermeasures

Circuit-level protection. *Power balancing* is used at circuit-level to minimize the side-channel leakage [15, 39, 60, 75, 82]. Others implement *gate-level masking* with complex logic gates [64, 69]. False-Key [89] uses a lightweight

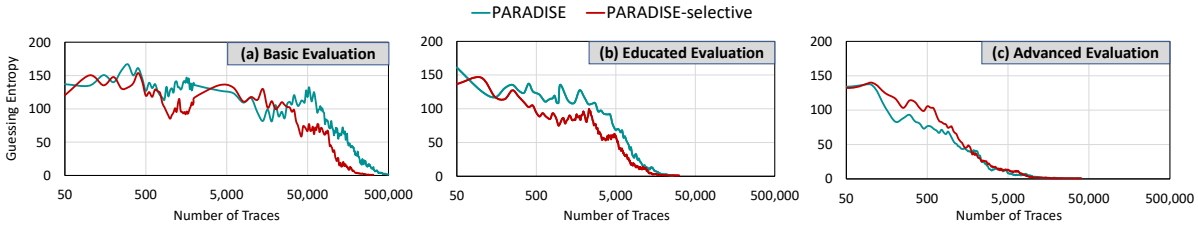

Fig. 19. Results of (a) basic, (b) educated, and (c) advanced security evaluation of PARADISE-selective compared to PARADISE. PARADISE-selective enables the protection only for secret-dependent regions.

Table 4. Comparison of existing power attack countermeasures for general-purpose processors.

	Countermeasure	Algorithm Agnostic	No Re-compile	Overheads [a] Area	Power	Performance	Security Evaluation [a] Basic	Educated	Advanced	Technique
In-Order	RIJID [3]	✓		2%	27%	30%	‡ [b]	– [c]	–	Random Code Injection
	Block Shuffler [11]	✓		2%	1.5%	0.7%	‡	–	–	Coarse Instr. Shuffling
	PARAM [7]	✓	✓	~20%	–	–	‡	–	–	Data Obfuscation
	Blinking [2]	✓	✓	–	–	270%	10 – 100×	10 – 100×	–	Power Hiding
Out-of-Order	random-iso-perf	✓	✓	~0%	0.8%	3.0%	12×	12×	5.5×	Fine Instr. Re-ordering
	random-iso-security	✓	✓	~0%	0.4%	12.0%	–	–	34×	Fine Instr. Re-ordering
	random-aggressive	✓	✓	~0%	0%	29.9%	122×	11×	38×	Fine Instr. Re-ordering
	PARADISE (*this work*)	✓	✓	0.8%	1.2%	3.2%	261×	12×	34×	Fine Instr. Re-ordering
	PARADISE + Masking	✓	✓	– [d]	– [d]	– [d]	–	–	62, 500×	Instr. Re-ordering+Mask
	baseline + Masking	✓	✓	– [d]	– [d]	– [d]	–	–	112×	Instr. Mask

[a] Values of other works are presented as reported. [b] '‡' for evaluation reported as negative and/or inconclusive tests that did not recover the key.
[c] '–' for imprecise reporting or absence of reporting of the information. [d] Masking overheads are beyond the scope of this work.

technique that combines power balancing and hardware masking. But, it is specific to AES engines. *Noise injection* and *power isolation* are techniques that hide the intermediate values of encryption operations [22, 34, 41, 42, 83, 84]. The authors of ANSI [22] combine these two techniques to implement a generic solution for encryption algorithms with negligible performance overhead. Most solutions at the circuit level incur high power and/or area overheads and are designed for encryption-specific hardware only.

Obfuscated execution. One class of mitigations for power analysis attacks is obfuscating the execution and data [55]. ARDPE [28] randomizes the data going on to a CGRA engine. However, it is only tailored to encryption engines. Blinking [2] uses a software controller to disconnect the core from the system during the leakiest moments of the execution but incurs a performance overhead of up to 2.7×. PARAM [7] investigates the leakiest modules of a RISC-V design and addresses them separately in a way that data will be in obfuscated form until they are processed. They require modifying the RTL code and de-obfuscating/obfuscating the data upon each access. Maya [65] is generic power obfuscation technique that uses formal control to keep the power close to the desired target function, but unlike PARADISE does not cover key extraction attacks.

Random code injection. RIJID [3] proposes a HW/SW co-design where the compiler marks the regions of the code that need protection, and the hardware injects (irrelevant) instructions at random intervals during their execution. This results in a 30% performance overhead.

Random shuffling. Block Shuffler [11] proposes a hardware/software co-design where the compiler detects independent blocks and inserts `shuffle` instructions that allow the processor to fetch instructions based on a random permutation of independent blocks (i.e., it implements a *coarse-grained instruction reordering* where independent blocks of the code are randomly reordered). However, PARADISE provides a *fine-grained instruction*

reordering strategy and we adopt such mechanism because of three main reasons: (1) coarse-grained reordering keeps the relative order of execution within a block of instructions which makes it easier for an adversary to detect the leaky regions of the block. (2) Detecting and partitioning a code into independent blocks for shuffling is usually algorithm-specific (e.g., prior work [11, 62] reorders the encryptions rounds of AES), and in addition, it requires software-level analysis and recompilation. On the other hand, Paradise provides a more generic solution without recompiling the binaries. Finally, (3) our fine-grained strategy to detect the slack and critical/non-critical instructions allows Paradise to have minimal performance overheads in a generic way (even for SPEC CPU2017 applications as representative benchmarks for performance analysis).

9.2 Security Evaluation of Prior Work

Table 4 also summarizes the reported security improvements of prior work and our evaluated designs with different types of adversaries. Note, that we compare the security benefits of each method to their corresponding unprotected baseline with respect to the same adversary.

The security evaluations of Blinking [2] falls into the educated category due to its SNR reduction technique. RIJID [3] and Block Shuffler [11] report inconclusive results and do not mount sufficient attacks to recover the secret keys. PARAM's [7] evaluation is classified as basic. However, they similarly report negative (inconclusive) results which are not sufficient to evaluate security, as they report the protected implementation is resistant with up to 1M traces. Moreover, the security of the obfuscation itself is not investigated.

10 Conclusion

In this work, we propose a general-purpose processor with a significantly improved level of security and always-on protection against power side-channel attacks, with little effect on application performance. We exploit the time between operand availability of critical instructions (*slack*) to create high-performance random schedules that obfuscate the execution with increased power noise. In addition, we introduce an advanced and comprehensive security evaluation model that complies with the highest security standards. Paradise offers a stronger security guarantee than demonstrated in previous works and improves security against advanced power analysis attacks by 556× when combined with Boolean Masking over a baseline only protected by masking, and 62, 500× over an unprotected baseline. Paradise introduces low performance, power, and area overheads of 3.2%, 1.2%, and 0.8% respectively.

References

[1] AES-128. Accessed 09-04-2023. https://github.com/openluopworld/aes_128.

[2] Alric Althoff, Joseph McMahan, Luis Vega, Scott Davidson, Timothy Sherwood, Michael Taylor, and Ryan Kastner. 2018. Hiding intermittent information leakage with architectural support for blinking. In *International Symposium on Computer Architecture (ISCA)*.

[3] Jude Angelo Ambrose, Roshan G. Ragel, and Sri Parameswaran. 2007. RIJID: Random Code Injection to Mask Power Analysis Based Side Channel Attacks. In *Design Automation Conference (DAC)*.

[4] Jude A. Ambrose, Roshan G. Ragel, and Sri Parameswaran. 2012. Randomized Instruction Injection to Counter Power Analysis Attacks. *ACM Transactions on Embedded Computing Systems* (2012).

[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. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* (2020).

[6] Cédric Archambeau, Eric Peeters, F-X Standaert, and J-J Quisquater. 2006. Template attacks in principal subspaces. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*.

[7] M. Arsath K F, V. Ganesan, R. Bodduna, and C. Rebeiro. 2020. PARAM: A Microprocessor Hardened for Power Side-Channel Attack Resistance. In *International Symposium on Hardware Oriented Security and Trust (HOST)*.

[8] Melissa Azouaoui et al. 2020. A systematic appraisal of side channel evaluation strategies. In *International Conference on Security Standardisation Research (SSR)*.

[9] Lejla Batina, Milena Djukanovic, Annelie Heuser, and Stjepan Picek. 2021. It Started with Templates: The Future of Profiling in Side-Channel Analysis. In *Security of Ubiquitous Computing Systems*.
[10] Lejla Batina, Benedikt Gierlichs, Emmanuel Prouff, Matthieu Rivain, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. 2011. Mutual information analysis: a comprehensive study. *Journal of Cryptology* (2011).
[11] Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne, and Wayne Burleson. 2012. An architecture-independent instruction shuffler to protect against side-channel attacks. *ACM Transactions on Architecture and Code Optimization* (2012).
[12] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. 2015. The SIMON and SPECK lightweight block ciphers. In *Design Automation Conference (DAC)*.
[13] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. 2020. Deep learning for side-channel analysis and introduction to ASCAD database. *Journal of Cryptographic Engineering* (2020).
[14] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*.
[15] Marco Bucci, Luca Giancane, Raimondo Luzzi, and Alessandro Trifiletti. 2006. Three-phase dual-rail pre-charge logic. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*.
[16] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. 2017. Convolutional neural networks with data augmentation against jitter-based countermeasures. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.
[17] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. 2002. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*.
[18] Md Hafizul Islam Chowdhuryy, Zhenkai Zhang, and Fan Yao. 2024. PowSpectre: Powering up speculation attacks with tsx-based replay. In *Asia Conference on Computer and Communications Security (AsiaCCS)*.
[19] Jeremy Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Pankaj Rohatgi, et al. 2013. Test vector leakage assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference (ICMC)*.
[20] Jean-Sébastien Coron. 2014. Higher order masking of look-up tables. In *International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*.
[21] Joan Daemen, Seth Hoffert, Michaël Peeters, G Van Assche, and R Van Keer. 2020. Xoodyak, a lightweight cryptographic scheme. *IACR Transactions on Symmetric Cryptology* (2020).
[22] D. Das, S. Maity, S. B. Nasir, S. Ghosh, A. Raychowdhury, and S. Sen. 2018. ASNI: Attenuated Signature Noise Injection for Low-Overhead Power Side-Channel Attack Immunity. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2018).
[23] Jesse De Meulemeester, Antoon Purnal, Lennert Wouters, Arthur Beckers, and Ingrid Verbauwhede. 2023. SpectrEM: Exploiting Electromagnetic Emanations During Transient Execution. In *USENIX Security Symposium*.
[24] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan. 2012. Side-channel vulnerability factor: A metric for measuring information leakage. In *International Symposium on Computer Architecture (ISCA)*.
[25] Whitfield Diffie and George Ledin. 2008. SMS4 encryption algorithm for wireless networks. *Cryptology ePrint Archive* (2008).
[26] François Durvaux and François-Xavier Standaert. 2016. From improved leakage detection to the detection of points of interests in leakage traces. In *International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*.
[27] B. Fields, S. Rubin, and R. Bodik. 2001. Focusing processor policies via critical-path prediction. In *International Symposium on Computer Architecture (ISCA)*.
[28] Wei GE, Shenghua CHEN, Benyu LIU, Min ZHU, and Bo LIU. 2020. A Power Analysis Attack Countermeasure Based on Random Data Path Execution For CGRA. *IEICE Transactions on Information and Systems* (2020).
[29] Barbara et al. Gigerl. 2021. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs.. In *USENIX Security Symposium*.
[30] Benjamin Jun Gilbert Goodwill, Josh Jaffe, Pankaj Rohatgi, et al. 2011. A testing methodology for side-channel resistance validation. In *NIST non-invasive attack testing workshop*.
[31] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*.
[32] Alagic Gorjan, Alperin-Sheriff Jacob, Apon Daniel, Cooper David, Dang Quynh, Kelsey John, Liu Yi-Kai, Miller Carl, Moody Dustin, Peralta Rene, Perlner Ray, Robinson Angela, and Smith-Tone Daniel. 2020. *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process.* National Institute of Standards and Technology (NIST).
[33] Vincent Grosso and François-Xavier Standaert. 2018. Masking proofs are tight and how to exploit it in security evaluations. In *International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*.
[34] Tim Güneysu and Amir Moradi. 2011. Generic side-channel countermeasures for reconfigurable devices. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*.
[35] Hendra Guntur, Jun Ishii, and Akashi Satoh. 2014. Side-channel attack user reference architecture board SAKURA-G. In *Global Conference on Consumer Electronics (GCCE)*.

[36] Miao He, Jungmin Park, Adib Nahiyan, Apostol Vassilev, Yier Jin, and Mark Tehranipoor. 2019. RTL-PSC: Automated power side-channel leakage assessment at register-transfer level. In *VLSI Test Symposium (VTS)*.
[37] Zhangqing He, Tianyong Ao, Meilin Wan, Kui Dai, and Xuecheng Zou. 2016. ERIST: An efficient randomized instruction insertion technique to counter side-channel attacks. *IAENG International Journal of Computer Science* (2016).
[38] Helmut Hlavacs, Thomas Treutner, Jean-Patrick Gelas, Laurent Lefevre, and Anne-Cecile Orgerie. 2011. Energy consumption side-channel attack at virtual machines in a cloud. In *International Conference on Dependable, Autonomic and Secure Computing (DASC)*.
[39] David D Hwang, Kris Tiri, Alireza Hodjat, B-C Lai, Shenglin Yang, Patrick Schaumont, and Ingrid Verbauwhede. 2006. AES-based security coprocessor IC in 0.18-*muhboxm* CMOS with resistance to differential power analysis side-channel attacks. *IEEE Journal of Solid-State Circuits* (2006).
[40] ISO. 2019. Information technology — Security techniques — Testing methods for the mitigation of non-invasive attack classes against cryptographic modules. In *International Organization for Standardization,*. https://www.iso.org/obp/ui/#iso:std:iso-iec:17825:ed-1:v1:en
[41] Monodeep Kar, Arvind Singh, Sanu Mathew, Anand Rajan, Vivek De, and Saibal Mukhopadhyay. 2017. 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)*.
[42] Monodeep Kar, Arvind Singh, Anand Rajan, Vivek De, and Saibal Mukhopadhyay. 2016. An integrated inductive VR with a 250MHz all-digital multisampled compensator and on-chip auto-tuning of coefficients in 130nm CMOS. In *European Solid-State Circuits Conference (ESSCIRC)*.
[43] 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 Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *International Symposium on Computer Architecture (ISCA)*.
[44] Keysight. Accessed 22-05-2023. Keysight InfiniiVision DSOX3104T Oscilloscope. https://www.keysight.com/sg/en/product/DSOX3104T.
[45] Diederik P Kingma and Jimmy Ba. 2017. Adam: a method for stochastic optimization (2014).
[46] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Annual International Cryptology Conference (CRYPTO)*.
[47] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2023. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In *USENIX Security Symposium*.
[48] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. 2018. Power to peep-all: Inference Attacks by Malicious Batteries on Mobile Devices. *Proc. Priv. Enhancing Technol.* (2018).
[49] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based power side-channel attacks on x86. In *IEEE Symposium on Security and Privacy (SP)*.
[50] Adam Malamy, Rajiv N Patel, and Norman M Hayes. 1994. Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature. US Patent 5,353,425.
[51] Stefan Mangard. 2004. Hardware countermeasures against DPA–a statistical analysis of their effectiveness. In *Cryptographers' Track at the RSA Conference*.
[52] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2008. *Power analysis attacks: Revealing the secrets of smart cards*. Springer Science & Business Media.
[53] George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* (2003).
[54] Macarena C Martínez-Rodríguez, Ignacio M Delgado-Lozano, and Billy Bob Brumley. 2021. SoK: Remote power analysis. In *International Conference on Availability, Reliability and Security (ARES)*.
[55] David May, Henk L Muller, and Nigel P Smart. 2001. Non-deterministic processors. In *Australasian Conference on Information Security and Privacy*.
[56] Daniel S. McFarlin, Charles Tucker, and Craig Zilles. 2013. Discerning the Dominant Out-of-Order Performance Advantage: Is It Speculation or Dynamism?. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[57] Turan Meltem Sönmez, McKay Kerry, Chang Donghoon, Çalık Çağdaş, Bassham Lawrence, Kang Jinkeon, and Kelsey John. 2021. *Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process*. National Institute of Standards and Technology (NIST).
[58] Thomas S Messerges, Ezzy A Dabbish, and Robert H Sloan. 1999. Investigations of Power Analysis Attacks on Smartcards. *Smartcard* (1999).
[59] Amir Moradi and François-Xavier Standaert. 2016. Moments-correlating DPA. In *Workshop on Theory of Implementation Security*.
[60] Maxime Nassar, Shivam Bhasin, Jean-Luc Danger, Guillaume Duc, and Sylvain Guilley. 2010. BCDL: A high speed balanced DPL for FPGA with global precharge and no early evaluation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.
[61] Siddika Berna Ors, Frank Gurkaynak, Elisabeth Oswald, and Bart Preneel. 2004. Power-analysis attack on an ASIC AES implementation. In *International Conference on Information Technology: Coding and Computing (ITCC)*.

[62] S. Patranabis, D. B. Roy, P. K. Vadnala, D. Mukhopadhyay, and S. Ghosh. 2016. Shuffling across rounds: A lightweight strategy to counter side-channel attacks. In *International Conference on Computer Design (ICCD)*.
[63] Stjepan Picek, Ioannis Petros Samiotis, Jaehun Kim, Annelie Heuser, Shivam Bhasin, and Axel Legay. 2018. On the performance of convolutional neural networks for side-channel analysis. In *International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*.
[64] Thomas Popp, Mario Kirschbaum, Thomas Zefferer, and Stefan Mangard. 2007. Evaluation of the masked logic style MDPL on a prototype chip. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*.
[65] Raghavendra Pothukuchi, Sweta Pothukuchi, Petros Voulgaris, Alex Schwing, and Josep Torrellas. 2021. Maya: Using Formal Control to Obfuscate Power Side Channels. In *International Conference on Computer Architecture (ISCA)*.
[66] Romain Poussier, François-Xavier Standaert, and Vincent Grosso. 2016. Simple key enumeration (and rank estimation) using histograms: an integrated approach. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*.
[67] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. 2009. Statistical analysis of second order differential power analysis. *IEEE Transactions on computers* (2009).
[68] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium*.
[69] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. 2015. Consolidating masking schemes. In *Annual Cryptology Conference*.
[70] Adi Shamir. 1979. How to share a secret. *Commun. ACM* (1979).
[71] Madura A Shelton, Łukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. 2021. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *Conference on Computer and Communications Security (CCS)*.
[72] Madura A Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. 2021. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In *Network and Distributed System Security Symposium (NDSS)*.
[73] Leslie N. Smith and Nicholay Topin. 2018. Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates. arXiv:1708.07120
[74] SMx Implementation Accessed 15-05-2024. https://github.com/NEWPLAN/SMx.
[75] Danil Sokolov, Julian Murphy, Alexander Bystrov, and Alexandre Yakovlev. 2005. Design and analysis of dual-rail circuits for security applications. *IEEE Trans. Comput.* (2005).
[76] François-Xavier Standaert. 2018. How (not) to use welch's t-test in side-channel security evaluations. In *International Conference on Smart Card Research and Advanced Applications (CARDIS)*.
[77] François-Xavier Standaert, Tal G Malkin, and Moti Yung. 2009. A unified framework for the analysis of side-channel key recovery attacks. In *International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*.
[78] Synopsys Design Compiler (DC). Accessed 15-05-2024. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.
[79] Synopsys PrimePower. Accessed 15-05-2024. https://www.synopsys.com/implementation-and-signoff/signoff/primepower.html.
[80] Synopsys VCS. Accessed 15-05-2024. https://www.synopsys.com/verification/simulation/vcs.html.
[81] The Berkeley Out-of-Order RISC-V Processor. Accessed 10-05-2024. https://github.com/riscv-boom/riscv-boom.
[82] Kris Tiri, Moonmoon Akmal, and Ingrid Verbauwhede. 2002. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *European Solid-State Circuits Conference (ESSCIRC)*.
[83] Carlos Tokunaga and David Blaauw. 2009. Secure AES engine with a local switched-capacitor current equalizer. In *International Solid-State Circuits Conference-Digest of Technical Papers*.
[84] C. Wang, M. Yan, Y. Cai, Q. Zhou, and J. Yang. 2017. Power Profile Equalizer: A Lightweight Countermeasure against Side-Channel Attack. In *International Conference on Computer Design (ICCD)*.
[85] Yiming Wen and Weize Yu. 2019. Machine learning-resistant pseudo-random number generator. *Electronics Letters* (2019).
[86] M Witteman, J Jaffe, and P Rohatgi. 2011. *Efficient side channel testing for public key algorithms: RSA case study.* Technical Report. Technical report, Cryptography Research.
[87] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. 2020. Open DNN box by power side-channel attack. *IEEE Transactions on Circuits and Systems II: Express Briefs* (2020).
[88] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. 2015. A study on power side channels on mobile devices. In *Asia-Pacific Symposium on Internetware*.
[89] Weize Yu and Selçuk Köse. 2017. A lightweight masked AES implementation for securing IoT against CPA attacks. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2017).
[90] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. 2020. Methodology for efficient CNN architectures in profiling attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES)* (2020).
[91] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. *Fourth Workshop on Computer Architecture Research with RISC-V* (2020).

Received 26 October 2023; revised 26 July 2024; accepted 6 September 2024