cryptography

MDPI

Article

Secure Implementation of RISC-V's Scalar Cryptography Extension Set

Asmaa Kassimi *, Abdullah Aljuffri, Christian Larmann, Said Hamdioui and Mottaqiallah Taouil *

Department of Computer Engineering, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2628 CD Delft, The Netherlands; a.a.m.aljuffri@tudelft.nl (A.A.); c.j.larmann@tudelf.nl (C.L.); s.hamdioui@tudelft.nl (S.H.)

* Correspondence: a.kassimi@tudelft.nl (A.K.); m.taouil@tudelft.nl (M.T.)

Abstract

Instruction Set Architecture (ISA) extensions, particularly scalar cryptography extensions (Zk), combine the performance advantages of hardware with the adaptability of software, enabling the direct and efficient execution of cryptographic functions within the processor pipeline. This integration eliminates the need to communicate with external cores, substantially reducing latency, power consumption, and hardware overhead, making it especially suitable for embedded systems with constrained resources. However, current scalar cryptography extension implementations remain vulnerable to physical threats, notably power side-channel attacks (PSCAs). These attacks allow adversaries to extract confidential information, such as secret keys, by analyzing the power consumption patterns of the hardware during operation. This paper presents an optimized and secure implementation of the RISC-V scalar Advanced Encryption Standard (AES) extension (Zkne/Zknd) using Domain-Oriented Masking (DOM) to mitigate first-order PSCAs. Our approach features optimized assembly implementations for partial rounds and key scheduling alongside pipeline-aware microarchitecture optimizations. We evaluated the security and performance of the proposed design using the Xilinx Artix7 FPGA platform. The results indicate that our design is side-channel-resistant while adding a very low area overhead of 0.39% to the full 32-bit CV32E40S RISC-V processor. Moreover, the performance overhead is zero when the extension-related instructions are properly scheduled.

Keywords: Advanced Encryption Standard (AES); scalar cryptography extensions; domain-oriented masking; side-channel attacks; RISC-V

Academic Editors: Jun Feng and Changqing Luo

Received: 17 December 2025
Revised: 7 January 2026
Accepted: 14 January 2026
Published: 17 January 2026

1. Introduction

With the exponential growth of Internet of Things (IoT) deployments—which is projected to exceed 125 billion devices by 2030 [1]—ensuring their protection from physical threats has become a significant challenge. Side-channel analysis (SCA) is among the most prominent classes of physical attacks, capable of compromising the security of IoT devices by exploiting unintended physical emissions such as power consumption, electromagnetic radiation, and timing variations. Although the National Institute of Standards and Technology (NIST) Cybersecurity Framework [2] only provides high-level guidance for managing cybersecurity risks, other NIST documents, such as NIST Special Publication 800-57 [3], it acknowledges the importance of protecting cryptographic modules against side-channel attacks. Furthermore, experimental research has shown that first-order side-channel attacks (e.g., Differential Power Attack (DPA) [4], Correlation Power Analysis (CPA) [5], and

Template Power Attacks (TPAs) [6]) on unprotected AES implementations can achieve high success rates with relatively few traces, emphasizing the critical need for robust countermeasures, especially in constrained environments like IoT RISC-V-based devices. Recent research has explored both hardware and software to secure AES implementations on RISC-V platforms against side-channel attacks (SCAs).

On the hardware side, Sinha et al. proposed PARAM, a side-channel-resistant RISC-V microprocessor that incorporates secure logic styles and obfuscated datapaths [7]. However, their evaluation was limited to a four-round Feistel network with an affine function, and no experimental validation was provided for a full AES software implementation. Therefore, the effectiveness of PARAM against practical AES-based SCAs remains unverified. Shaout et al. introduced AES-RV, a low-latency hardware AES accelerator featuring custom RISC-V instructions [8]. Although their solution improves performance, it depends on non-standard instructions, potentially limiting compatibility with existing RISC-V compilers and toolchains. On the software side, Sajadi et al. explored lightweight countermeasures such as dummy instruction insertion and software masking for the Secure-Ibex core [9], though these techniques incurred noticeable performance penalties. Cui et al. further improved masking efficiency by proposing a software-oriented instruction set extension (ISE) for field multiplication in GF(2^8), integrated with an RISC-V-compatible hardware accelerator [9]. Nevertheless, their design also relies on custom hardware and non-standard instructions. To the best of our knowledge, there is currently no side-channel-resistant implementation based on the standard RISC-V AES scalar cryptographic extensions (Zkne/Zknd) [10].

This paper introduces a secure AES module compatible with RISC-V's Zkne/Zknd extensions [10], achieving minimal area and execution time overhead while being resistant against first-order SCA attacks like CPA and TPA. The module is secured using Domain-Oriented Masking (DOM) [11]. In summary, the contributions of this paper are as follows:

- The proposal of an optimized unsecure AES module through shared Sbox logic (i.e., logic optimized both for encryption and decryption) for RISC-V's Zkne/Zknd extensions.
- The proposal of a DOM-protected AES module against SCAs for RISC-V's Zkne/Zknd extensions with minimal area overhead.
- Assembly-level optimizations for partial-round operations and key scheduling to realize zero execution overhead.
- Empirical security validation using evaluation-style (i.e., CPA and TPA) and conformance-style (i.e., Test Vector Leakage Assessment (TVLA) and Signal-to-Noise Ratio (SNR)) testing to ensure compliance with NIST's SCA resilience guidelines [12].
- Comprehensive evaluation of area and power overhead and comparisons with the state of the art.

This paper is organized as follows: Section 2 discusses the related work. Section 3 introduces the background on AES, RISC-V cryptography extensions, and'SCAs. Section 4 presents our proposed optimization of the AES design for the Zkne/Zknd extensions and its DOM-based secure implementation. Section 5 evaluates the area and performance overhead, and its security using TVLA, SNR, and key rank analysis. Section 6 discusses the results. Finally, Section 7 concludes this paper.

2. Related Work

In this section, we describe related work that focuses on AES implementations with eight-bit datapaths, i.e., implementations in which the number of used Sboxes is reduced from 16 to 1 to significantly save area and power. The RISC-V AES scalar cryptographic extensions Zkne/Zknd are also based on an eight-bit datapath. The authors of [13] performed the MixColumns operation after 4 sequential Sbox operations were computed.

In [14], the authors processed the AES state one byte per clock cycle and directly performed a partial MixColumns operation and optimized the register file. In [15,16], the authors created sub-modules that are shared in the datapath of both the encryption and decryption. Other eight-bit datapath implementations [17–19] are very similar to the one proposed in [13]. Most of these works neglect decryption, and none of them have integrated any security or countermeasures against PSCAs. Adding countermeasures is extremely expensive. Moradi et al. [17] showed that the area overhead is approximately 300% for a 128-bit encryption-only datapath with threshold implementation (TI). Similarly, with DOM, the area overhead is 200% [11].

In the domain of RISC-V scalar cryptography extensions, Tran et al [20] integrated multiple extensions (Zbkb, Zbkx, Zknh, Zknd, Zkne) into the CORE-V Wally processor using shared Sbox logic and unified datapaths, achieving substantial speedups (5× AES, 2× SHA-256) and reduced code size, but without addressing side-channel security. Similarly, Marshall et al [21] implemented RISC-V AES scalar extensions, but omitted protection mechanisms and used separate encryption/decryption Sboxes, roughly doubling the area compared to those of the shared designs. To the best of our knowledge, there is currently no side-channel-resistant implementation based on the standard RISC-V AES scalar cryptographic extensions (Zkne/Zknd) [10] that maintains low area and performance overhead. Hence, there is still a demand for low-cost secure scalar extensions.

3. Background

This section provides a brief background on the AES algorithm, RISC-V Zkne/Zknd instruction set, and SCAs.

3.1. AES Algorithm

AES is a symmetric encryption algorithm widely used in cybersecurity. It ensures data confidentiality by protecting data through encryption and decryption. Figure 1 shows the different modules it consists of. Data are encrypted/decrypted in fixed 128-bit blocks using a key length of 128, 192, or 256 bits. The selected key length determines the number of cryptographic rounds: N_r = 10, 12, or 14, respectively, for the three different key lengths. Round keys are generated from the Key expansion module for each round. The 128-bit input message (or plaintext) can be represented by 16 bytes, stored in a 4×4 array referred to as a state. Each round of encryption includes four primary modules that operate on the input message: SubBytes, ShiftRows, MixColumns, and AddRoundKey. After each operation, the 4 × 4 array is updated and a new state is stored. The Key expansion module and the four primary modules are explained below in more detail.

Key expansion: Let N_r be the number of rounds (10, 12, or 14 for 128/192/256-bit keys) and N_k denote the key size in 32-bit words (i.e., N_k = 4, 6, or 8). The Key expansion module generates $4(N_r + 1)$ 32-bit partial round keys from the original cipher key. An example is shown in Figure 2 for a 128-bit key. The expanded key array $W[0, 1, \ldots, 4(N_r + 1)]$ is constructed as follows:

- *Initialization*: Initialize the first N_k words with the cipher key:

$$W[0, 1, \ldots, N_k - 1] = \text{cipher key}$$

- *Iterative Expansion*: For words $i \geq N_k$, compute

$$W[i] = \begin{cases} W_{i-N_k} \oplus \text{Sbox}(\text{SLR}(W_{i-1})) \oplus \text{rc}_{i/N_k}, & i \equiv 0 \pmod{N_k} \\ W_{i-N_k} \oplus \text{Sbox}(W_{i-1}), & N_k > 6 \wedge i \equiv 4 \pmod{N_k} \\ W_{i-N_k} \oplus W_{i-1}, & \text{otherwise} \end{cases} \quad (1)$$

where

- Sbox(x) applies the AES Sbox substitution to byte x.
- SLR(x) performs a byte circular left shift on a 32-bit word. For example, $\text{SLR}([b_0, b_1, b_2, b_3]) = [b_1, b_2, b_3, b_0]$.
- rc[x] contains a 32-bit word that includes the round constant. For example, for Round 2, rc[2] = (0x02, 0x00, 0x00, 0x00). All the round constants are defined in Table 1. The last two operations, SLR and rc, are part of the F function in Figure 2, and their implementation is shown in Figure 3.

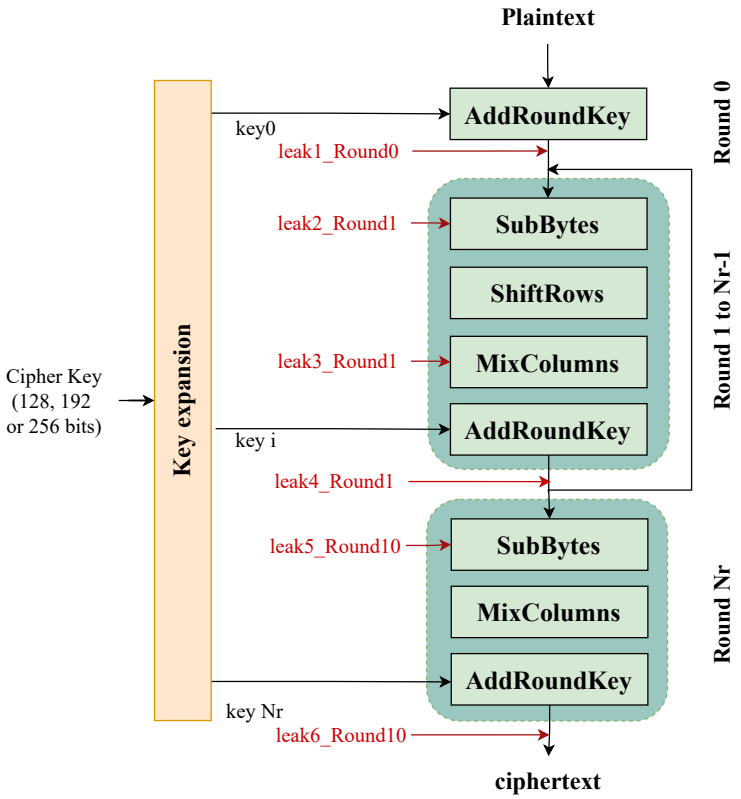


Figure 1. AES Encryption and Decryption

AddRoundKey: This module performs a bitwise XOR between the state array and the round key obtained from the Key expansion module.

SubBytes: This module (also referred to as Sbox) is the only non-linear module in AES and plays a crucial role in thwarting differential and linear cryptanalysis [11]. Each byte in the state array is substituted with another byte. The substitution is based on a multiplicative inversion in the Galois Field GF(2^8) combined with an affine transformation [22].

ShiftRows (SR): In this module, the state matrix is updated by cyclically shifting the second, third, and fourth rows by one, two, and three bytes to the left, respectively.

MixColumns (MC): This module performs a modular polynomial multiplication in Galois Field GF(2^8) on each column of the state array in Equation (2). Given a column s of the state matrix, MC computes the following matrix vector products:

$$\text{MixColumns}(s) = \begin{cases} \begin{bmatrix} 03 & 01 & 01 & 02 \\ 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \end{bmatrix} s & \text{(Encryption)} \\ \begin{bmatrix} 0B & 0D & 09 & 0E \\ 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \end{bmatrix} s & \text{(Decryption)} \end{cases} \quad (2)$$

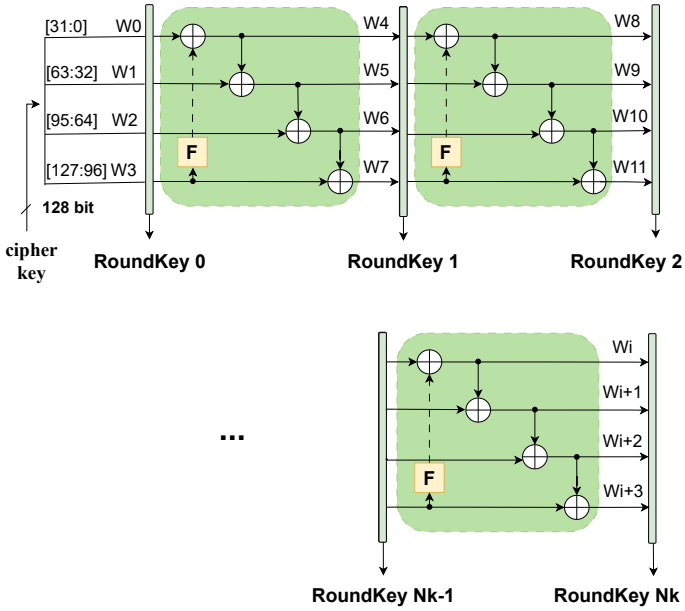


Figure 2. Key Expansion

The decryption operation consists of the inverse operations of these modules: they are called InvSubBytes, InvShiftRows, and InvMixColumns.

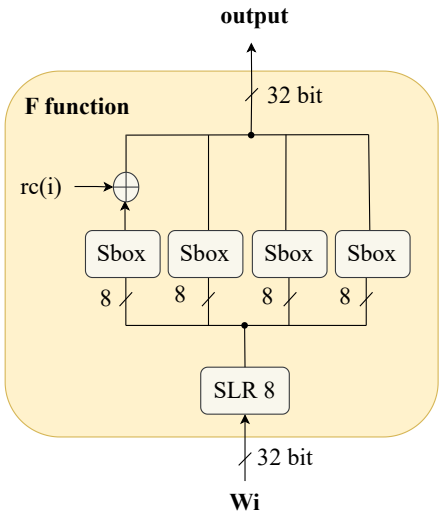


Figure 3. F Function in Key Expansion Module.

Table 1. Round Constants (rc[i]).

i	1	2	3	4	5	6	7	8	9	10
rc[i]	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1B	0x36

3.2. AES RISC-V Zkne/Zknd Instruction Set

The RISC-V scalar cryptography extensions provide hardware acceleration for AES operations through dedicated instructions (aes32esmi, aes32esi, aes32dsmi, and aes32dsi), as shown in Table 2. The aes32esmi and aes32esi are used for the middle rounds (MRs) and final rounds of the encryption, and similarly, aes32dsmi and aes32dsi for decryption. The instruction format for aes32dsmi can be found in Figure 4. It is part of the R-format type [23] and has four arguments: rs1 initially contains the partial-round key, and rs2, part of the input state for that round. The bs bits are used to select which byte is being computed, as only a single Sbox is available. The result contains an Sbox operation, SR (based on bs), and a partial MC operation and is stored in rd. In successive instructions, rs1 will contain partially computed MC results that were previously stored in rd. The final round is handled in a similar manner using aes32esi instructions which only contain the forward Sbox and SR transformations (i.e., no MC).

Table 2. AES ZKne/Zknd Instructions.

Instruction	**Formula**
aes32esmi	$\texttt{rd} = \texttt{rs1} \oplus \texttt{Mix}(\texttt{Sbox}(\texttt{SR}(\texttt{rs2})))$
aes32esi	$\texttt{rd} = \texttt{rs1} \oplus \texttt{Sbox}(\texttt{SR}(\texttt{rs2}))$
aes32dsmi	$\texttt{rd} = \texttt{rs1} \oplus \texttt{InvMix}(\texttt{InvSbox}(\texttt{InvSR}(\texttt{rs2})))$
aes32dsi	$\texttt{rd} = \texttt{rs1} \oplus \texttt{InvSbox}(\texttt{InvSR}(\texttt{rs2}))$

Mnemonic

aes32esmi rd, rs1, rs2, bs

Encoding

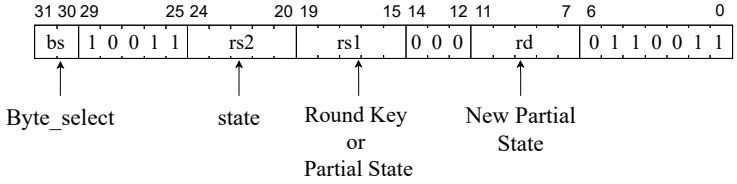


Figure 4. Zkne aes32esmi Middle-Round Encryption Instruction.

Similarly, the decryption operation is computed using aes32dsmi that applies inverse Sbox (InvSbox) and partial inverse MixColumns (InvMC) for middle rounds, and aes32dsi for the final round. To execute a complete AES round, Zkne/Zknd instructions must be repeated 4 times to iteratively calculate the 4 bytes of the middle-round's output.

An example is provided in Figure 5 and Table 3. It shows how four aes32esmi instructions are iteratively used to compute a full AES middle round. As shown in the figure, we have transformed the state array from column orientation to word orientation and reversed the byte order to have the LSB bytes on the right side to facilitate the processing (for example, key operations).

Table 3. Zkne Encryption Round Using aes32esmi Instructions.

Instruction	State Value	Initial t0 (rs1)	Final t0 (rd)
aes32esmi t0, t0, a4, 0	$3 \cdot \texttt{Sbox}(6c) \oplus 0x17$ $1 \cdot \texttt{Sbox}(6c) \oplus 0xf6$ $1 \cdot \texttt{Sbox}(6c) \oplus 0xf4$ $2 \cdot \texttt{Sbox}(6c) \oplus 0xa0$	17fefaa0	e7aeaa00
aes32esmi t0, t0, a5, 1	$1 \cdot \texttt{Sbox}(1b) \oplus 3 \cdot \texttt{Sbox}(6c) \oplus 0x17$ $1 \cdot \texttt{Sbox}(1b) \oplus 1 \cdot \texttt{Sbox}(6c) \oplus 0xf6$ $2 \cdot \texttt{Sbox}(1b) \oplus 1 \cdot \texttt{Sbox}(6c) \oplus 0xf4$ $3 \cdot \texttt{Sbox}(1b) \oplus 2 \cdot \texttt{Sbox}(6c) \oplus 0xa0$	e7aeaa00	4801efeaa
aes32esmi t0, t0, a6, 2	$1 \cdot \texttt{Sbox}(e8) \oplus 1 \cdot \texttt{Sbox}(1b) \oplus 3 \cdot \texttt{Sbox}(6c) \oplus 0x17$ $2 \cdot \texttt{Sbox}(e8) \oplus 1 \cdot \texttt{Sbox}(1b) \oplus 1 \cdot \texttt{Sbox}(6c) \oplus 0xf6$ $3 \cdot \texttt{Sbox}(e8) \oplus 2 \cdot \texttt{Sbox}(1b) \oplus 1 \cdot \texttt{Sbox}(6c) \oplus 0xf4$ $1 \cdot \texttt{Sbox}(e8) \oplus 3 \cdot \texttt{Sbox}(1b) \oplus 2 \cdot \texttt{Sbox}(6c) \oplus 0xa0$	4801efeaa	d32c5971
aes32esmi t0, t0, a7, 3	$2 \cdot \texttt{Sbox}(45) \oplus 1 \cdot \texttt{Sbox}(e8) \oplus 1 \cdot \texttt{Sbox}(1b) \oplus 3 \cdot \texttt{Sbox}(6c) \oplus 0x17$ $3 \cdot \texttt{Sbox}(45) \oplus 2 \cdot \texttt{Sbox}(e8) \oplus 1 \cdot \texttt{Sbox}(1b) \oplus 1 \cdot \texttt{Sbox}(6c) \oplus 0xf6$ $1 \cdot \texttt{Sbox}(45) \oplus 3 \cdot \texttt{Sbox}(e8) \oplus 2 \cdot \texttt{Sbox}(1b) \oplus 1 \cdot \texttt{Sbox}(6c) \oplus 0xf4$ $1 \cdot \texttt{Sbox}(45) \oplus 1 \cdot \texttt{Sbox}(e8) \oplus 3 \cdot \texttt{Sbox}(1b) \oplus 2 \cdot \texttt{Sbox}(6c) \oplus 0xa0$	d32c5971	0f9e371f

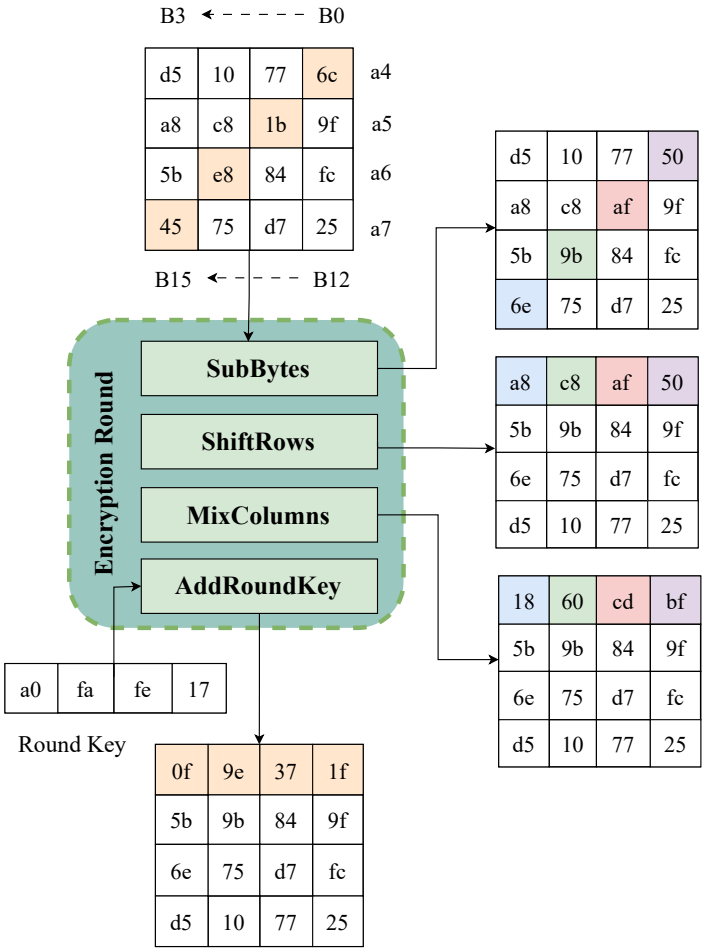


Figure 5. Zkne Scalar Encryption Round.

The four yellow highlighted cells at the input of the SubBytes operation and output after AddRoundKey are targeted, as they together are involved in the MC operation together. Note that due to the transformation, the SR operation is now performed vertically instead of horizontally. Let us assume that the partial round key for the four targeted output cells is equal to 0x17fefaa0 and initially stored in register *t0*. Furthermore, the round inputs are stored in registers a4–a7 (e.g., a4 = 0xd5f0776c and a5 = 0xa8c81b9f). Table 3 shows how four instructions are applied to obtain the round result. The state value in the

table (i.e., 0x17f6f4a0) represents the intermediate result of plaintext ⊕ round_key from Round 0, which is distinct from the round key value held in *t0*. The active calculations are shown in Figure 5, with the colors matching the byte colors:

- In the first instruction, where bs = 0, selects a4's byte 0x6c, applies Sbox and partial MixColumns, and then performs XORs with t0 (which contains the partial-round key) to yield *0xe7aeaa00*.
- Secondly, when bs = 1, it processes state input 0x1b which is stored in the second byte of a5, performs Sbox and MC. As the MC matrix contains four columns, which are rotated each byte, the same MC is performed as for bs = 0 but the output is byte-rotated by one byte and subsequently XORed with the partial result of the first instruction stored in t0; rotating the MC result is beneficial as the second column of the MC matrix is equal to the first one when one byte is rotated, as can be seen in Equation (2). This observation also applies to the remaining MC matrix columns.
- Thirdly, when (bs = 2), it similarly processes state input 0xe8, which is the third byte of a6. Then, the MC result is byte-rotated by two bytes.
- Finally, when bs = 3, byte 0x45 is processed. Then, the MC result is byte-rotated by three bytes.

After four instructions, *t0* holds the final column state *0x0f9e371f*, completing a full round.

The decryption, where the ciphertext is the initial state, is performed in a similar manner using aes32dsmi. However, it contains the inverse Sbox, SR, and MC operations. By operating on 8 bits of the input and using the standard 32-bit registers, the design avoids wide datapaths, prioritizing area efficiency over throughput. The RISC-V scalar extensions [10] guarantee constant execution time (as all computations are performed using registers), and'hence prevent simple timing side-channel attacks. However, they lack built-in protection mechanisms against power or electromagnetic analysis and hence, require additional micro-architectural hardening for secure implementations.

3.3. Power Side-Channel Attacks (SCAs)

Modern power SCAs exploit statistical relationships between a device's power consumption and secret cryptographic operations. In this paper, we consider three popular attack methods: DPA [4], CPA [5], and TPA [6]. DPA and CPA rely on simpler statistical models and are non-profiled attacks, while TPA employs advanced multivariate profiling.

DPA statistically extracts data-dependent leakage by averaging power traces over multiple executions. Let T_i denote a power trace and D_i denote a bit of intermediate data (e.g., the LSB output bit of the S-Box). Based on a guessed key k^*, and the provided plaintext, the output of the S-Box can be computed. Subsequently, the N traces are partitioned into subsets S_0 and S_1 based on the value of D_i [4]:

Note that when a wrong key is assumed, the collected traces in S_0 and S_1 have no meaning and the difference between the average between them is small (due to random assignment). However, when the key is correctly guessed, all traces in S_0 will have the LSB output bit of the Sbox equal to 0 and similarly all traces in S_1 equal to 1. Taking the difference of the averages of both sets will reveal the slightly systematic difference between them. This is expressed by the following equation [4]:

$$\Delta(k^*) = \frac{1}{|S_1|} \sum_{i \in S_1} T_i - \frac{1}{|S_0|} \sum_{i \in S_0} T_i \tag{3}$$

where the maximum value of $\Delta(k^*)$ determines the presumed correct key guess k, as it has the highest probability were power leakage is correlated with D_i.

CPA [5] is a powerful side-channel technique that exploits the linear relationship between a device's power consumption and hypothetical power models derived from intermediate cryptographic values. It refines DPA using Pearson's correlation coefficient to compare a hypothetical power model $H(k^*)$ (e.g., Hamming weight) with measured traces T:

$$\rho(k^*) = \frac{\mathrm{Cov}(H(k^*), T)}{\sigma_{H(k^*)}\sigma_T} \tag{4}$$

The attacker computes $\rho(k^*)$ for all key guesses k^*. The key guess with the highest correlation $|\rho(k^*)|$ is presumed to be correct. Popular power or leakage models are the Hamming weight (HW) and Hamming distance. In this paper, we consider several intermediate attack points, as shown in Figure 1. The attack points are denoted with the different leak keywords such as leak1_Round0. In total, we consider six leakage attack points.

TPA [6] uses multivariate statistics to profile devices. It consists of two phases: profiling and extraction phase. During the profiling phase, power traces are modeled as multivariate Gaussian distributions for each class c (e.g., intermediate value):

$$p(\mathbf{t}|c) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}_c|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{t}-\boldsymbol{\mu}_c)^\top \mathbf{\Sigma}_c^{-1}(\mathbf{t}-\boldsymbol{\mu}_c)\right) \tag{5}$$

Here, $\boldsymbol{\mu}_c$ and $\mathbf{\Sigma}_c$ are the mean and covariance of the traces for class c. In the extraction phase, the templates created during the profiling phase are used to guess the key by computing the likelihood of the observed traces for each template. Bayes' theorem identifies the most likely c given a target trace $\mathbf{t}$:

$$\hat{c} = \arg\max_c \; p(c|\mathbf{t}) = \arg\max_c \; p(\mathbf{t}|c)p(c) \tag{6}$$

4. Secure Scalar Cryptography Extension

This section presents our proposed implementation approach for the optimized unsecure AES and its protected version using DOM [11]. We begin with the motivation behind our approach and then provide a detailed explanation of its design, implementation, and RISC-V integration.

4.1. Motivation

The increasing prevalence of side-channel attacks on cryptographic implementations requires robust hardware-level countermeasures. Previous AES implementations like [21,24] typically suffer from two critical limitations: (1) conditional logic for encryption/decryption modes introduces timing and power leakage channels, and (2) unprotected Sboxes expose intermediate values, which are prime targets for DPA, CPA, and TPA. To address these challenges, we first focus on the optimization of the AES design presented in [21] to reduce area and power consumption for resource-constrained IoT devices. This unsecure and unoptimized design is shown in Figure 6. By unifying the datapath and simplifying the logic for encryption and decryption, we eliminate redundant logic (e.g., no separate Sboxes for forward/inverse operations) and approximately reduce the gate count by 22%. A high-level schematic of these optimizations is shown in Figure 7. Both unsecure designs (i.e., the non-optimized [21] and optimized versions) will be discussed further in the next subsection. Secondly, we propose a protected implementation based on DOM but tailored for RISC-V's Zkne/Zknd extensions. This protected design also benefits from resource sharing, while enabling secure encryption and decryption. Protection is provided by integrating masking operations (through the introduction of random numbers)

at every layer of the Sbox, thereby guaranteeing protection against first-order SCAs without compromising on performance. Random numbers are generated with a linear feedback shift register (LFSR). In each cycle, new fresh random numbers are used, thus improving resistance against higher-order side-channel attacks. Although this approach increases the complexity of such attacks, it does not make them theoretically impossible.

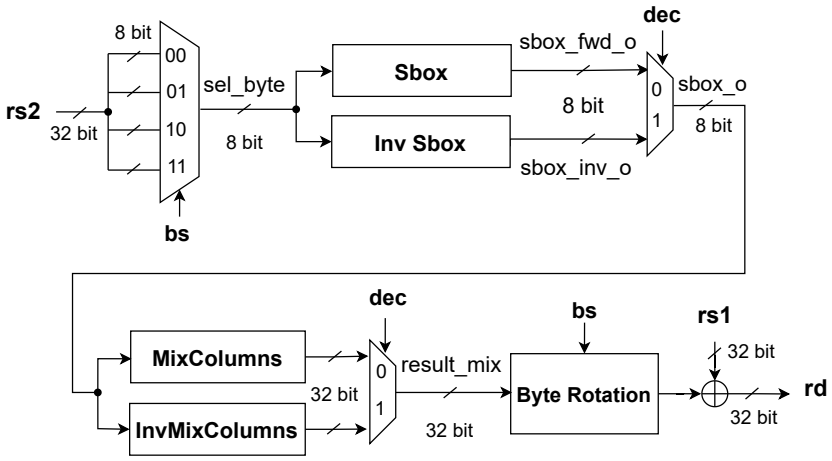


Figure 6. Unsecure Unoptimized AES Partial Round [21].

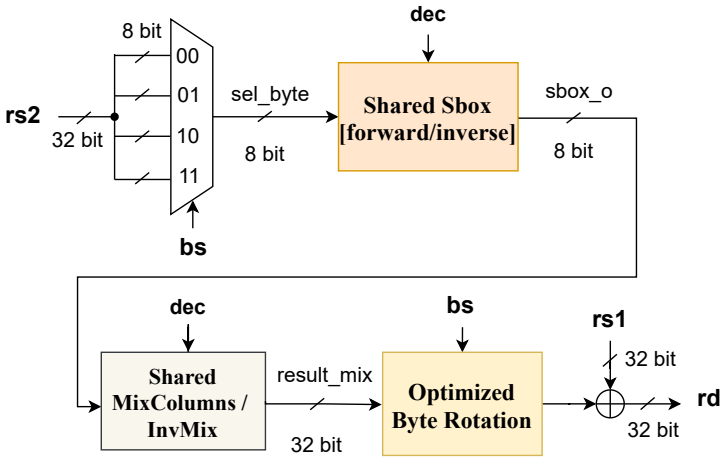


Figure 7. Unsecure Optimized AES Partial Round

4.2. Design and Implementation of Proposed AES Optimization

This section describes our optimizations with respect to the design proposed in [21] to enhance both its efficiency and security, while adhering to RISC-V's scalar cryptography extensions [10]. As shown in Figure 7, we created a shared Sbox architecture, which allows a single module to perform both forward and inverse Sbox operations depending whether an encryption or decryption instruction is being run (which is denoted by the signal dec in the figure). More details on the shared Sbox architecture design are shown in Figure 8. The design contains three layers based on the design in [25]: linear top and bottom layers and a shared non-linear middle layer. The top linear layer for encryption applies the forward affine transformation, while its inverse counterpart implements the inverse affine transform for decryption [25]. Similarly, the bottom linear layer for encryption applies the complementary forward affine transformation to compress intermediate value back to the final eight-bit encrypted output. Its decryption counterpart implements the inverse affine transform to reconstruct the plaintext byte [25]. The non-linear middle layer (which implements a GF inversion) is shared between them, eliminating redundant hardware

and reducing the Sbox area by 22%. Optionally, we allow conditional instantiation of the inverse Sbox. In case the decryption is disabled, 32% of the Sbox area can be reduced in encryption-only configurations.

The MixColumns operation is further optimized by reducing the number of performed GF multiplications (*xtimeN_opt*) by reusing intermediate terms (e.g., $x_2 = xtime2(a)$); this reduces the combinatorial delay by 19% and gate count by 18%.

Finally, handshake logic is simplified and rotation operations are transformed into a barrel-shifter structure, reducing the logic by 12%. Altogether, these optimizations achieve a 22% overall gate reduction while maintaining single-cycle throughput. An in-depth explanation of the AES module optimizations presented in Figure 7 are detailed further below.

1. **Shared Forward and Inverse Sbox:** The original design [21] instantiates both forward and inverse Sboxes separately and uses a multiplexer to select between Sbox outputs (see Figure 6):

$$\text{Sbox\_o} = \begin{cases} \text{inv Sbox(sel\_byte)}, & \text{if dec=1} \\ \text{Sbox(sel\_byte)}, & \text{otherwise} \end{cases} \tag{7}$$

However, our optimized AES Sbox (Figure 7) uses linear top and bottom layers to apply either the forward or inverse affine transforms while sharing the non-linear middle layer between encryption and decryption[25]:

$$\begin{aligned} \text{sbox\_fwd\_out} &= \text{Bottom\_layer}_{\text{fwd}}(GF_{inv}(\text{Top\_layer}_{\text{fwd}}(sel\_byte))) \\ \text{sbox\_inv\_out} &= \text{Bottom\_layer}_{\text{inv}}(GF_{inv}(\text{Top\_layer}_{\text{inv}}(sel\_byte))) \end{aligned} \tag{8}$$

The shared non-linear middle layer in Equation (9) computes a GF inversion in GF(2^8) (denoted by GF_{inv}) using the multiplicative inverse with irreducible polynomial $\text{P(x)} = \text{x}^8 + \text{x}^4 + \text{x}^3 + \text{x} + 1$:

$$GF_{inv}(x) = x^{-1} \mod \text{P(x)} \tag{9}$$

Overall, the shared Sbox outputs can be expressed as follows:

$$\text{sbox\_o} = \begin{cases} \text{sbox\_fwd\_out} & \text{(encryption)} \\ \text{sbox\_inv\_out} & \text{(decryption)} \end{cases} \tag{10}$$

2. **MixColumns Optimization:** The reduced GF multiplication logic is shared within the MixColumns/InvMixColumns block, as shown in Figure 7, reducing hardware for both operations. The MixColumns operation in AES, which depends on the inputs after shiftRows, requires finite field multiplications with fixed constants (e.g., 2, 3, 9, 11). The original design in [21] computed these constants dynamically using nested calls to the *xtime2* function. The *xtime2* function [26] is defined as

$$xtime2(a) = \begin{cases} (a \ll 1) & \text{if } a_7 = 0 \\ (a \ll 1) \oplus 0x1B & \text{otherwise} \end{cases} \tag{11}$$

For constants like $11 = 1 + 2 + 8$, the original code is calculated using the following equation:

$$a \cdot 11 = a \oplus xtime2(a) \oplus xtime2^3(a) \tag{12}$$

where $xtime2^3(\text{a}) = xtime2(xtime2(xtime2(\text{a})))$.

The optimized version simplifies the intermediate term by computing $xtime2(a)$ once and reusing it as follows:

$$x_2 = xtime2(a) \tag{13}$$

$$x_4 = xtime2(x_2) \tag{14}$$

$$x_8 = xtime2(x_4) \tag{15}$$

$$xtimeN\_opt(a, b) = \bigoplus_{i=0}^{3} b_i \cdot x_{2^i} \tag{16}$$

This saves one xtime2() operation.

3. **Byte Rotation Optimization:** The original rotation logic uses explicit bitwise operations for each *byte_sel* operation, which is synthesized into a 4:1 multiplexer. The optimized version uses a barrel-shifter-like structure (where bs represents the byte selected):

$$\text{rotated} = \text{result} \gg (8 \times bs) \mid \text{result} \ll (32 - 8 \times bs) \tag{17}$$

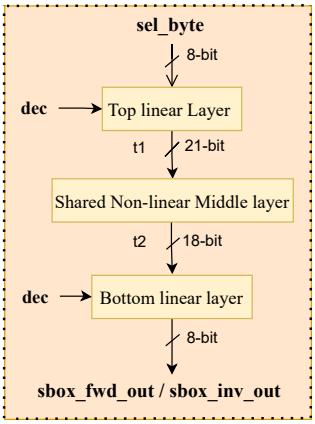


Figure 8. Shared Forward/Inverse Sbox Design.

4.3. Design and Implementation of Proposed DOM Countermeasure

The DOM design safeguards against first-order side-channel attacks by splitting sensitive variables into two shares and injecting fresh randomness between linear and non-linear computational layers. As the countermeasure must be compliant to the instruction definition, prior to computing the partial round, we first generate the shares and combine them at the end to produce rd, as illustrated in Figure 9. It is not possible to store rd as 2 shares as there is no room in the instruction for it. Instead, each instruction generates its own new fresh shares, which are combined after the computation. The design of the DOM-protected AES accelerator features a shared pipeline that processes both encryption and decryption operations, which is similar to that of the unsecure but optimized design. By sharing critical resources such as Sboxes and MixColumns modules, the design minimizes area overhead while preserving security guarantees. The design employs a four-stage pipeline that synchronizes the processing of the shares without having leakage due to temporary glitches in different paths. Random numbers are injected at strategic points inside the Sbox to disrupt statistical dependencies between shares. The optimized shared Sbox implementation, depicted in Figure 10, shares the logic for forward (AES) and inverse (AES^{-1}) operations. Four key optimizations drive our design's efficiency. They are explained below.

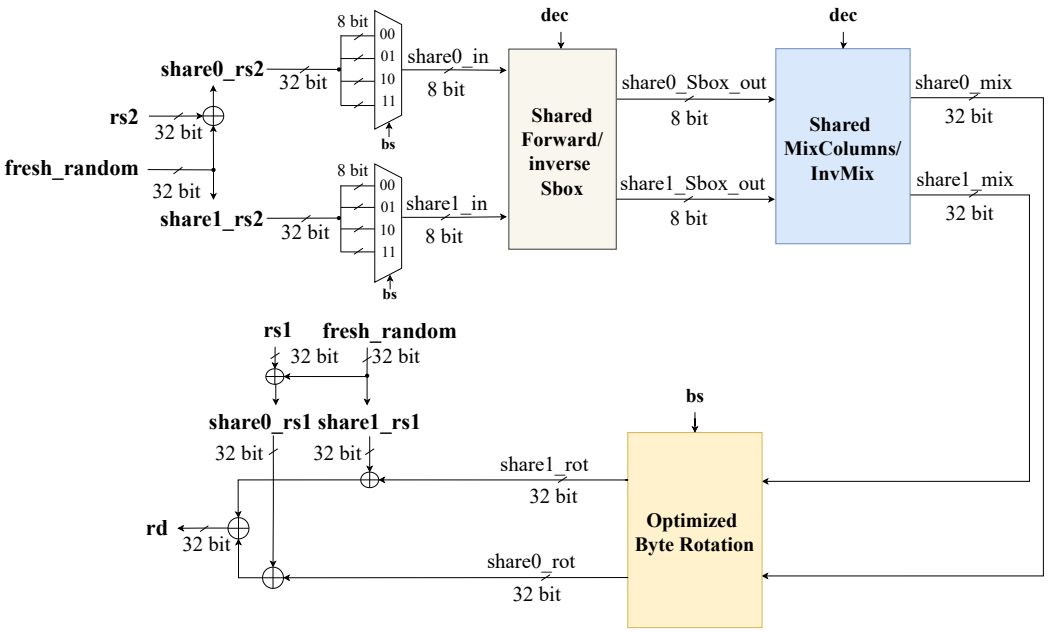


Figure 9. Proposed Design for DOM Encryption and Decryption.

Random Number Generation: The DOM design splits sensitive rs1 and rs2 inputs into two shares using fresh random numbers:

$$\begin{aligned} \text{share0\_rs1} &= \text{rs1} \oplus \text{fresh\_random\_number} \\ \text{share1\_rs1} &= \text{fresh\_random\_number} \\ \text{share0\_rs2} &= \text{rs2} \oplus \text{fresh\_random\_number} \\ \text{share1\_rs2} &= \text{fresh\_random\_number} \end{aligned} \tag{18}$$

Ideally, a true random number generator is used to generate the fresh random numbers. However, as it might be costly to generate each cycle of fresh numbers, we propose using a 32-bit linear feedback shift register (LFSR). The LFSR is used to generate a random number each clock cycle. It has a period of $(2^{32} - 1)$ cycles, i.e., it requires that amount of cycles before the same random number is repeated. This ensures that the random state does not repeat within any practical measurement window, preventing statistical correlation between masks. To prevent potential leakage, we initialize the seed with an actual random number each time an encryption or decryption takes place. This design ensures resilience against side-channel attacks, including vertical attacks based on trace collection [27].

Layered Randomness Injection and Shared Encryption/Decryption Sbox Logic: The fresh random number (fresh_random_number) generated by the LFSR module is injected between the linear top layer and non-linear middle layer, as well as between the middle layer and linear bottom layer of the shared DOM Sbox module, as shown in Figure 10. The Sbox implementation is similar to the unprotected optimized design, but two shares are computed instead. The inputs and outputs of this module can also be seen in Figure 9, where the computed masks are highlighted in red boxes and present the following:

$$\begin{aligned} \text{mask\_t1\_0} &= \text{share0\_t1} \oplus \text{r\_top} \\ \text{mask\_t1\_1} &= \text{share1\_t1} \oplus \text{r\_top} \\ \text{mask\_t2\_0} &= \text{share0\_t2} \oplus \text{r\_mid} \\ \text{mask\_t2\_1} &= \text{share1\_t2} \oplus \text{r\_mid} \end{aligned} \tag{19}$$

Equation (19) provides distinct masks for forward and inverse operations, effectively isolating cross-domain leakage inside the Sbox. They include the following security features: (1) mathematical independence between shares and (2) fresh random numbers breaking statistical dependencies and preventing vertical attacks [27]. The design ensures first-order side-channel resistance by having all intermediates as uniform random variables while preserving cryptographic correctness.

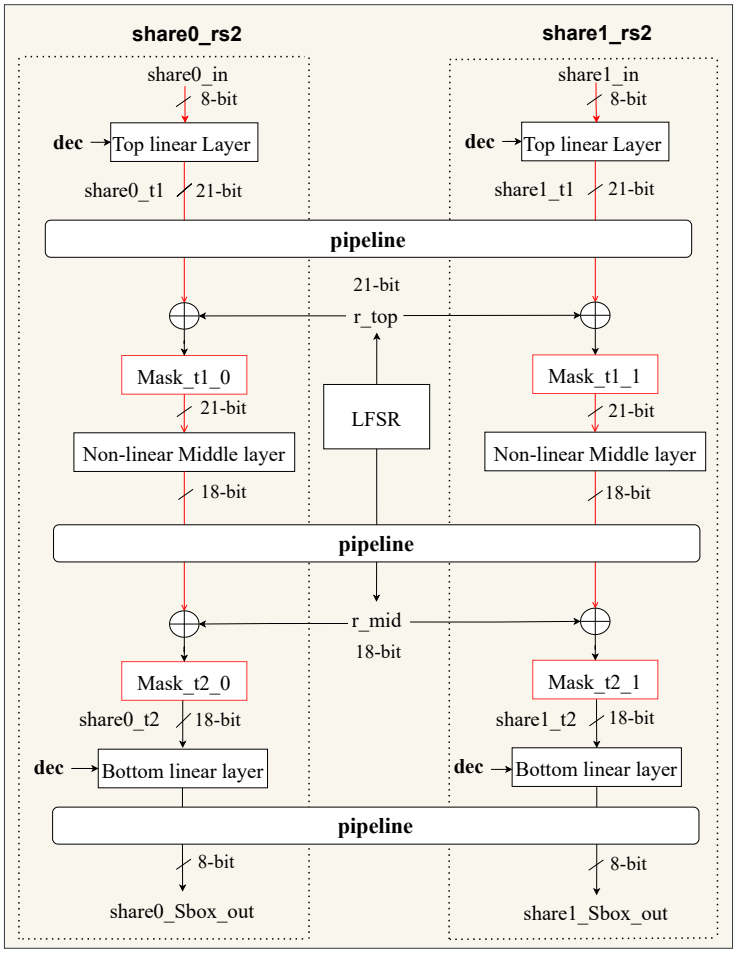


Figure 10. Proposed 1st-order DOM Shared Sbox Module

Shared MixColumns Logic: The linear MixColumns transformation operates on individual shares in GF(2^8) through two key arithmetic optimizations. First, the shared MixColumns forward/inverse logic is integrated into a single hardware module to compute both AES encryption (using coefficients 0x03, 0x02, 0x01) and decryption (using coefficients 0x0B, 0x0D, 0x09, 0x0E) using Equation (2) controlled by the dec control signal. This eliminates duplicate circuitry by dynamically selecting Galois Field GF(2^8) multiplication factors through multiplexers, reducing the area overhead by 15%.

Second, the lightweight xtimeN function replaces traditional lookup tables with combinatorial logic for GF(2^8) multiplication. The core operation uses the xtime2 primitive in the same way as previously explained for the unsecure optimized AES, as shown in Equation (16).

The shared MixColumns module is applied to each share separately (share0_sbox_out and share1_sbox_out). Since MixColumns is a linear operation, processing shares individually preserves the correctness based on the following equation:

$$\text{Mix\_Col}(s_0 \oplus s_1) = \text{Mix\_Col}(s_0) \oplus \text{Mix\_Col}(s_1) \tag{20}$$

Prevention of Leakage Due to Glitches: The four-stage DOM pipeline is balanced to prevent leakages due to glitches, which sometimes occurs when the computational paths are not synchronized. For example, leakage can occur when the random numbers arrive later to the XOR gates in Equation (18) than the actual signal (i.e., rs1 or rs2). However, due to forced synchronization using pipelines, this is not possible in the design.

4.4. Integrating Zkne/Zknd Scalar Cryptography into the CV32E40S Pipeline

The AES extension unit is integrated into the CV32E40S 5-stage pipeline as a functional unit in the execute stage. To decode the instructions, the decoder in the instruction/decode stage was extended to recognize the four AES scalar instructions (aes32esmi, aes32esi, aes32dsmi, aes32dsi). It generates control signals for operation type, byte select, and source operands. Note that no modifications to the load/store unit were required, as AES instructions operate solely on register operands. To provide the AES module with random numbers in each cycle, a 32-bit LFSR is used.

5. Experimental Results

This section presents the experimental setup, and the security and performance evaluations of the unprotected and DOM-protected AES implementations.

5.1. Experimental Setup

We evaluated the security and efficiency of the unprotected design proposed in [21], our optimized version, and the DOM-protected design by subjecting them to comprehensive testing. To quantify security, we performed DPA, CPA, and TPA using Python 3.6.13 and Python's SCALib library 0.3.4, and NumPy 1.19.2 against the unprotected and DOM-protected AES designs. We further validated the leakage resilience of the three designs using empirical methods such as TVLA and SNR. We generated all traces on a ChipWhisperer CW305 board manufactured by NewAE Technology Inc., Halifax, Nova Scotia, Canada [28], which contains an Artix-7. To measure the overhead and power consumption, we implemented the three designs in TSMC CMOS 40nm technology using Cadence Genus [29]. The power consumption was estimated in Genus using Value Change Dump (VCD) files that contain thousands of encryptions and decryptions.

5.2. Security Evaluation

In this section, we analyze the security of the designs using the potential leakage points shown in Figure 1. They are listed again in Table 4 with their leakage models. Using DPA, CPA, and TPA, we targeted the critical operations: AddRoundKey, SubBytes, and MixColumns. The number of traces required for successful attacks for the unsecure unoptimized design [21] is shown in Table 5. For example, at point Leak1_Round0 with the leakage model Hamming weight (HW) applied to the plaintext (pt) XORed with the key (i.e., HW[pt ^ key), 300 to 500 traces are needed to attack all key bytes with CPA, while only 10 to 50 traces are needed when TPA is used. The range denotes the minimum and maximum number of traces needed to perform a successful attack on the key bytes. Note that some bytes of the key are easier to attack. We added a specific attack point for the Zkne/Zknd instructions called Leak4_round1. This potential leakage point attacks the partial round computed following the instructions. We further split this attack point into two cases: Leak4a, where we attack a single byte of the instructions, and Leak4b, where all four bytes are attacked.

Table 4. Different leakage points in AES Algorithm.

Attack Point	Leakage Model
Leak1_Round0	HW[pt ^ key]
Leak2_Round1	HW[Sbox[pt ^ key]]
Leak3_Round1	HW[(Sbox[pt ^ key] * 2) & 0xFF]
Leak4_Round1	HW[mc_out ^ Round_key]
Leak5_Round10	HW[Sbox(ct ^ key)]
Leak6_Round10	HW[cipher ^ key]

Table 5. DPA, CPA, and TPA Results for All Attack Points.

Attack Point	DPA Traces Min-Max	CPA Traces Min-Max	Template Traces Min-Max
Leak1_Round0	50–200	300–500	10–50
Leak2_Round1	100–300	500–4000	20–50
Leak3_Round1	300–500	500–5000	30–100
Leak4a_Round1	300–500	500–4000	20–50
Leak4b_Round1	500–1000	1000–10,000	50–200
Leak5_Round10	50–200	500–4000	10–50
Leak6_Round10	50–200	300–500	10–50

The results in Table 5 show critical vulnerabilities in the unprotected AES implementation. For some attack points, only 50 traces were needed to attack all key bytes. Similarly, the results of our optimized unprotected design have been omitted. These results highlight the critical need for countermeasures like DOM to prevent exploitable power side-channel leakage.

The DOM-protected implementation (see Figure 9) demonstrates significant resilience against CPA attacks. The key rank analysis results in Figure 11 shows that the correct key candidates for all tested bytes (the figure shows only bytes 0 to 5) remain at high ranks (>100) even with 10,000 traces, indicating no leakage. The key rank analysis across traces confirms that DOM effectively prevents power side-channel leakage. Similar results were obtained for TPA attacks, as shown in Figure 12, which shows the partial guessing entropy (PGE). The unprotected design is very vulnerable as it requires very few traces for a successful attack. In contrast, the DOM-protected design maintains a high PGE (>100 at 10,000 traces), demonstrating robust resistance. Residual leakage, visible as minor PGE fluctuations, likely arises from noise and the mismatches between the attack's unmasked Hamming weight model and DOM's masked implementation.

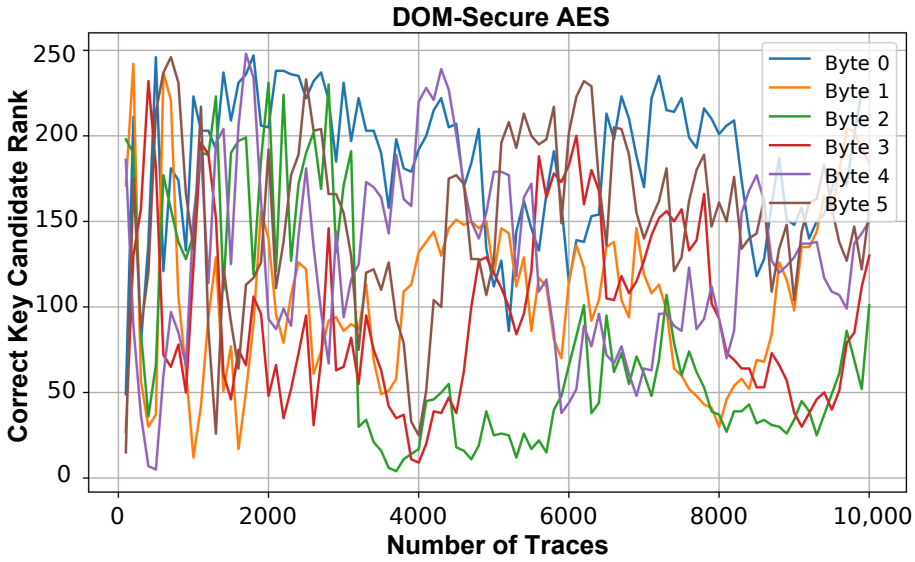


Figure 11. Key Rank Analysis of DOM-Protected AES Designs.

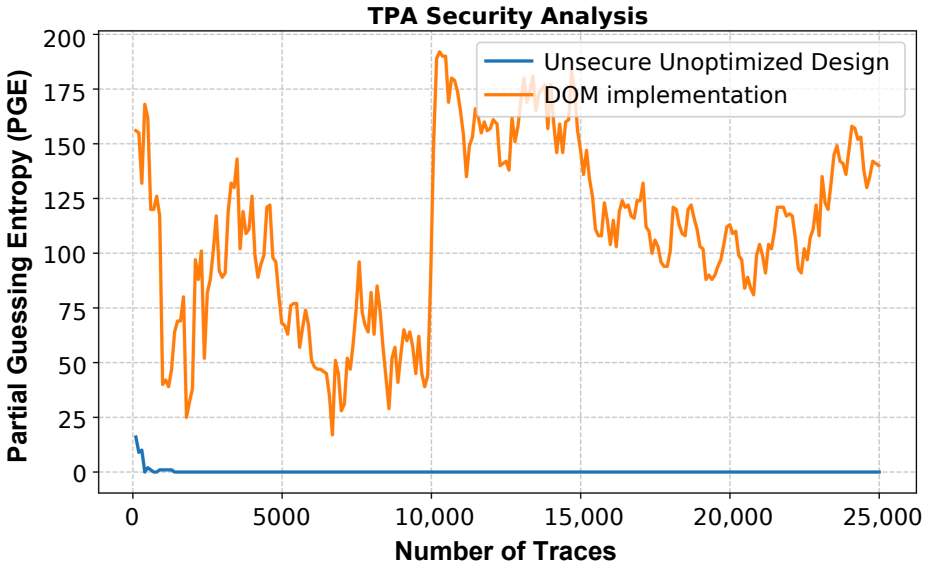
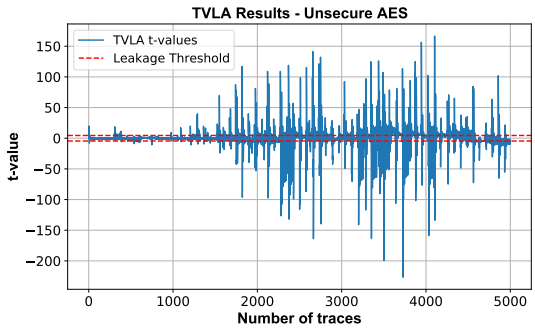
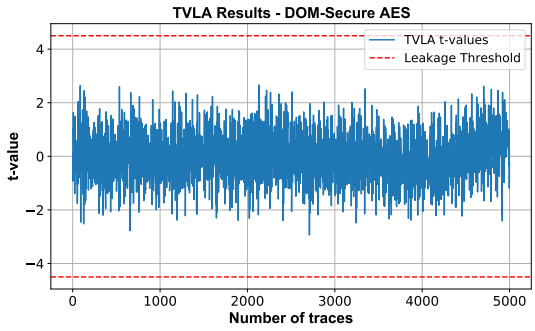


Figure 12. TPA Analysis of Unsecure [21] and DOM-Secure AES Designs.

Moreover, we assessed the side-channel leakage using: (1) TVLA [30] to detect information leakage through power trace analysis, and (2) SNR [31] to quantify potential exploitability by measuring key-dependent signal strength against operational noise. The unsecure AES implementation [21] has a lot of leakage as shown in Figure 13a. Through the application of the standard leakage threshold ($|t| > 4.5$) [30], the unprotected design shows widespread signal leakage across all sample points (0–5000 index range), with t-values significantly exceeding the threshold. This is validated by its SNR profile [31], shown in Figure 14a, where values peak at 0.25, significantly exceeding the 0.1 threshold [32]. This indicates a strong correlation between power consumption and sensitive operations.

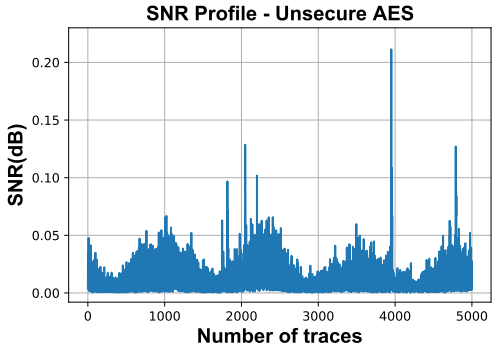


(**a**)

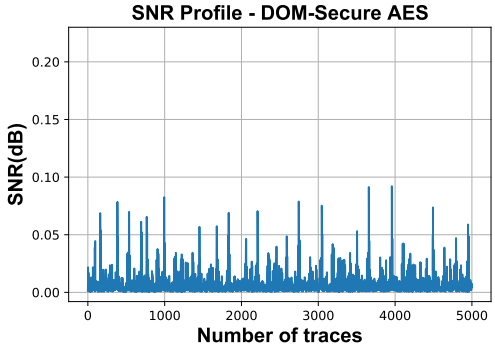


(**b**)

Figure 13. General TVLA Analysis of Unsecure [21] and DOM-Secure AES Designs. (**a**) TVLA Analysis for Unsecure AES [21]; (**b**) TVLA Analysis for DOM-Secure AES.



(**a**)



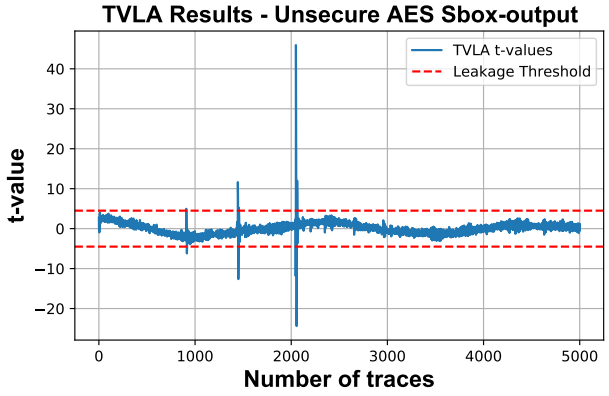
(**b**)

Figure 14. SNR Analysis of Unsecure [21] and DOM-Secure AES designs. (**a**) Unprotected AES [21] Signal-to-Noise-Ratio; (**b**) DOM-Protected AES Signal-to-Noise-Ratio.

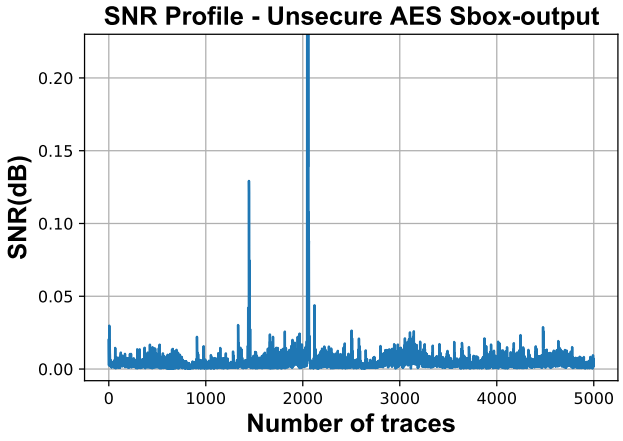
To confirm that the leakage comes from the Sbox, we also applied the TVLA and SNR analysis method proposed in [33], which is tailored to leakage in Sboxes. Again, the unprotected Sbox shows alarming TVLA spikes ($\|t\| > 20$) in Figure 15a and a high SNR > 0.15 in Figure 15b, making the secret key extraction trivial.

In contrast, the DOM-secured design demonstrates significantly improved resilience. Under identical TVLA thresholds, leakage amplitudes are drastically reduced (see Figure 13b), with t-values below the threshold of 4.5. The SNR profile shown in Figure 14b confirms this, showing maximum values below 0.10 and reduced trace deviations, which reflect DOM's effectiveness in obscuring data-dependent patterns.

Overall, the DOM-secured AES achieves a strong balance between security and performance and prevents side-channel risks inherent to the unsecured design.



(**a**)



(**b**)

Figure 15. TVLA and SNR Analysis of Unsecure AES Sbox Output [21]. (**a**) TVLA Analysis of Unprotected AES Sbox-Output in [21]; (**b**) AES Sbox-output Signal-to-Noise-Ratio.

5.3. Performance Evaluation

Table 6 shows the area and power comparison of the three designs.We synthesized the designs for a clock frequency of 100 MHz and 200 MHz, and for all three cases, the synthesis results are the same, indicating that the three designs can run at higher frequencies. The area of our secure DOM-based design is 12.8% smaller than that of the unsecure unoptimized design presented in [21], and only 37% larger than that of our unsecure optimized design. The DOM-based design only adds 0.39% to the complete CV32E40S RISC-V processor. Also, the power consumption of our optimized designs is much lower (about 50%). However,

in the case of DOM, four cycles are required to complete the instruction, and hence, the total energy required is higher for the secure implementation.

Table 6. Standalone Unsecure and DOM-Secure AES Overhead Analysis Using TSMC 40nm.

Design	Freq. (MHz)	Area (μm^2)	Area Ratio	Power (μW)
Unsecure 1-Sbox unoptimized AES [21]	100–200	966.024	1	18.404
Unsecure 1-Sbox optimized AES	100–200	530.098	0.55	8.864
1-Sbox Secure DOM	100–200	841.666	0.87	6.407

As shown in Table 7, the DOM-protected design exhibits nearly identical FPGA resource usage to the unsecure 1-Sbox AES baseline, with only a slight reduction in LUT consumption (12.40% vs. 13.10%) and marginal differences in FF usage, while all other resources (BRAMs, DSPs, IOs, BUFGs) remain unchanged.

Table 7. Comparison of FPGA Resource Utilization: DOM-Protected vs. Unsecure 1-Sbox Unoptimized AES [21].

Resource	DOM-Protected			Unsecure 1-Sbox AES [21]		
	Util.	Avail.	Util.%	Util.	Avail.	Util.%
LUTs	7863	63,400	12.40%	8304	63,400	13.10%
FFs	4985	126,800	3.93%	4947	126,800	3.90%
BRAMs	29	135	21.48%	29	135	21.48%
DSPs	7	240	2.92%	7	240	2.92%
IOs	44	170	25.88%	44	170	25.88%
BUFG	2	32	6.25%	2	32	6.25%

The drawback of the secure DOM implementation is that it requires four cycles to complete a single instruction. The pipeline stall logic was extended to account for the four-cycle DOM latency, hence the dependency between instructions; see, for example, the four instructions in the first column of Table 3, which will lead to a significantly longer execution time. However, this drawback can be fully mitigated when instructions are reordered. This will be shown in the following subsection. The DOM implementation is fully pipelined, allowing for the start of a new operation each cycle. The full pipeline was needed anyway to protect against leakages due to temporary glitches.

5.4. AES Assembly Optimization

To understand how to optimize the assembly instructions, we first have to explain how the instructions are used to calculate a complete round. For brevity, we only demonstrate this for the middle rounds (MRs) using aes32esmi and ignore register initialization. Note that the final round works in a similar manner but uses the aes32esi instruction instead.

A complete middle round (MR) is shown in Listing 1. It first loads the 128-bit round key and stores it in four registers (U0 to U3). The Key expansion module generates the round keys; it performs its computations separately and is addressed later. Thereafter, four groups with four instructions each are used to complete the entire round. An example of a single group of four instructions is shown in Table 3. Together, the four instructions compute four bytes of the middle round. Hence, 4×4 instructions are needed to compute the 16 output bytes.

For the unsecure designs, pipeline stalling does not occur when the aes32esmi instructions are executed, as the results are directly available in the next cycle when forwarding is used. However, it takes four cycles to complete the DOM operation, and hence, a secure implementation will have a large execution overhead as the pipeline has to be stalled three cycles each time a dependency occurs. To eliminate this overhead, we propose reordering the instructions, as shown in Listing 2. Each time we compute a partial round belonging to a certain group of four instructions (which generates the four output bytes of the round), we switch the next cycle to a partial round instruction of the next group. This helps us completely hide the latency.

Listing 1. Conventional MR.

```
// Load Round Key\\
    lw U0, 16(RK) \\
    lw U1, 20(RK)\\
    lw U2, 24(RK)\\
    lw U3, 28(RK)\\

    aes32esmi U0, U0, T0, 0 \\
    aes32esmi U0, U0, T1, 1\\
    aes32esmi U0, U0, T2, 2\\
    aes32esmi U0, U0, T3, 3\\

    aes32esmi U1, U1, T1, 0\\
    aes32esmi U1, U1, T2, 1\\
    aes32esmi U1, U1, T3, 2\\
    aes32esmi U1, U1, T0, 3\\

    aes32esmi U2, U2, T2, 0\\
    aes32esmi U2, U2, T3, 1\\
    aes32esmi U2, U2, T0, 2\\
    aes32esmi U2, U2, T1, 3\\

    aes32esmi U3, U3, T3, 0\\
    aes32esmi U3, U3, T0, 1\\
    aes32esmi U3, U3, T1, 2\\
    aes32esmi U3, U3, T2, 3 \\
```

The Key expansion module is shown in Figure 2 and presented in Listing 3. The top part defines the different round constants. The important part is the loop (presented by .aes_128_enc_ks_l0). During the execution of the loop, it stores the round key, computes the F-function in Figure 3 (consisting of a rotate operation, four Sbox operations, and a round constant), and finally performs three XOR operations at the end (see also Figure 2). The F-function is primarily computed using four aes32esi instructions. Unfortunately, these instructions are also affected by pipeline stalling, and hence, we also propose to reorder them by combining middle rounds with the key expansion.

Listing 4 shows the optimized Key expansion module mixed with the middle round operations. During the execution of Round i, we already compute the key for the next round. This ensures that we have no performance penalties, despite having a latency of four cycles to compute the AES-related instructions. Another benefit of computing in this manner is that the round keys do not have to be stored as they are directly used in the next round.

Listing 2. Optimized MR.

```
// Load Round Key
   lw U0, 16(RK)
   lw U1, 20(RK)
   lw U2, 24(RK)
   lw U3, 28(RK)

   aes32esmi U0, U0, T0, 0
   aes32esmi U1, U1, T1, 0
   aes32esmi U2, U2, T2, 0
   aes32esmi U3, U3, T3, 0

   aes32esmi U0, U0, T1, 1
   aes32esmi U1, U1, T2, 1
   aes32esmi U2, U2, T3, 1
   aes32esmi U3, U3, T0, 1

   aes32esmi U0, U0, T2, 2
   aes32esmi U1, U1, T3, 2
   aes32esmi U2, U2, T0, 2
   aes32esmi U3, U3, T1, 2

   aes32esmi U0, U0, T3, 3
   aes32esmi U1, U1, T0, 3
   aes32esmi U2, U2, T1, 3
   aes32esmi U3, U3, T2, 3
```

Listing 3. Key Expansion logic.

```
aes_round_const:
    .byte 0x01, 0x02, 0x04, 0x08, 0x10
    .byte 0x20, 0x40, 0x80, 0x1b, 0x36

AES_LOAD_STATE C0,C1,C2,C3,CK,t0,t1,t2,t3

   mv RKP, RK
   addi RKE, RK, 160
   la RCP, aes_round_const

.aes_128_enc_ks_l0: // Loop~start

    sw C0, 0(RKP) // rkp[0] = a2
    sw C1, 4(RKP) // rkp[1] = a3
    sw C2, 8(RKP) // rkp[2] = a4
    sw C3, 12(RKP) // rkp[3] = a5

    addi RKP, RKP, 16 // increment~rkp

    lbu RCT, 0(RCP) // Load rc byte
    addi RCP, RCP, 1 // Increment rc byte
    xor C0, C0, RCT

    ROR32I T1, RCT, C3, 8
    aes32esi C0, C0, T1, 0
    aes32esi C0, C0, T1, 1
    aes32esi C0, C0, T1, 2
    aes32esi C0, C0, T1, 3

    xor C1, C1, C0
    xor C2, C2, C1
    xor C3, C3, C2
```

https://doi.org/10.3390/cryptography10010006

Listing 4. Complete AES Round with Key Scheduling.

```
.aes_128_enc_ks_l0: // Loop start
    lbu RCT, 0(RCP)
    addi RCP, RCP, 1
    xor C0, C0, RCT

    ROR32I TMP, RCT, C3, 8
    aes32esi C0, C0, TMP, 0

    aes32esmi U0, U0, T0, 0
    aes32esmi U1, U1, T1, 0
    aes32esmi U2, U2, T2, 0
    aes32esmi U3, U3, T3, 0

    aes32esi C0, C0, TMP, 1
    aes32esmi U0, U0, T1, 1
    aes32esmi U1, U1, T2, 1
    aes32esmi U2, U2, T3, 1
    aes32esmi U3, U3, T0, 1

    aes32esi C0, C0, TMP, 2
    aes32esmi U0, U0, T2, 2
    aes32esmi U1, U1, T3, 2
    aes32esmi U2, U2, T0, 2
    aes32esmi U3, U3, T1, 2

    aes32esi C0, C0, TMP, 3
    aes32esmi U0, U0, T3, 3
    aes32esmi U1, U1, T0, 3
    aes32esmi U2, U2, T1, 3
    aes32esmi U3, U3, T2, 3

    // Finish key expansion
    xor C1, C1, C0
    xor C2, C2, C1
    xor C3, C3, C2
```

6. Discussion

In this paper, we introduced a highly efficient secure AES accelerator optimized for low power and a compact area, without compromising on performance. Nevertheless, several aspects require further attention. They are discussed below:

Area Overhead vs. Security: Using techniques such as resource sharing, targeted module optimizations, and instruction rescheduling, our design achieves significant improvements over the only existing solution, reducing area by up to 45% and power consumption by 51% compared to the unoptimized implementation [21]. Furthermore, we extended these optimizations to develop a lightweight DOM-based side-channel countermeasure, resulting in a secure Sbox design that still has a 12.9% lower area than the original proposed unsecured AES module. Additionally, our DOM implementation incurs only a 37% area overhead compared to the unprotected optimized design; this is significantly lower than the typical 200–300% overhead in masked implementations [11,17,34]. Although the DOM-protected design significantly reduces power leakage (evidenced by 60% lower SNR values and 3.15× reduced leakage amplitude in TVLA tests), residual leakage near

detection thresholds persists, leaving a potential vulnerability window for advanced or new attacks.

Larger Datapaths: The current work focuses on a single Sbox configurations (i.e., an eight-bit datapath), limiting throughput for latency-critical applications. It is possible to explore 32-bit or 64-bit datapaths. However, maximum performance will be challenging to achieve. For example, when using a 32-bit datapath based on four available Sboxes, the challenge would be to select the correct four bytes each round, as illustrated in Figure 5. If four additional instructions are used for byte selection, all gains are lost. Hence, smarter schemes need to be considered.

Higher-Order Attacks: To evaluate the security of our DOM implementation, we also have to consider higher-order side-channel attacks that target multiple attack points simultaneously. A practical example is the second-order CPA proposed in [35], where the combination of leakages from multiple points is analyzed. For example, in attacks on masked AES implementations, attackers typically combine leakages from the masked Sbox output [36]. Furthermore, deep learning-based profiled attacks [37] have successfully broken high-order DOM implementations. These methods are particularly effective against hardware implementations, where temporary glitches leak information [11]. In part, our future work aims to improve the resilience of the DOM design against these higher-order attacks.

7. Conclusions

In this study, we conducted a comprehensive security assessment of an innovative AES accelerator for RISC-V's Zkne/Zknd extensions. The accelerator was optimized for low power, compact area, and execution time overhead. Our approach features three key innovations: a shared Sbox architecture enabling combined encryption/decryption operations, a lightweight DOM-based countermeasure providing side-channel protection with minimal area impact, and assembly-level optimizations for efficient partial-round processing.

The designs were rigorously evaluated through empirical TVLA testing and side-channel vulnerability analysis, confirming compliance with the National Institute of Science and Technology (NIST)'s SCA resilience guidelines. The results demonstrate that our DOM-protected implementation not only maintains equivalent first-order security compared to conventional approaches but also achieves significant area reduction. This paper demonstrates that a low-cost security overhead for IoT-constrained devices is feasible.

Author Contributions: Conceptualization, A.K., A.A. and M.T.; Methodology, A.K., A.A., C.L., S.H. and M.T.; Software, A.K., A.A., C.L., S.H. and M.T.; Validation, A.K., A.A., C.L., S.H. and M.T.; Formal analysis, A.K., A.A. and M.T.; Investigation, A.K., A.A. and M.T.; Resources, A.K., A.A. and M.T.; Data curation, A.K. and M.T.; Writing—original draft, A.K.; Writing—review & editing, A.K., A.A., S.H. and M.T.; Visualization, A.K., A.A. and M.T.; Supervision, A.A., S.H. and M.T.; Project administration, S.H. and M.T.; Funding acquisition, S.H. and M.T. All authors have read and agreed to the published version of the manuscript. .

Funding: The project is supported by the Chips JU and its members including top-up funding by the Netherlands Enterprise Agency under grant agreement No. 101112282.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding authors.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. IHS Markit. The Internet of Things: A Movement, Not a Market. 2017. Available online: https://cdn.ihs.com/www/pdf/IoT_ebook.pdf (accessed on 1 June 2025).
2. National Institute of Standards and Technology. *The NIST Cybersecurity Framework (CSF) 2.0*; NIST Cybersecurity White Paper (CSWP) NIST CSWP 29; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2024. https://doi.org/10.6028/NIST.CSWP.29.
3. National Institute of Standards and Technology. *Recommendation for Key Management—Part 1: General*; Technical Report NIST SP 800-57 Part 1 Revision 5, NIST; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2020. https://doi.org/10.6028/NIST.SP.800-57pt1r5.
4. Kocher, P.C.; Jaffe, J.; Jun, B. Differential Power Analysis. In *Proceedings of the Advances in Cryptology—CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, CA, USA, 15–19 August 1999*; Proceedings; Wiener, M.J., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1666, pp. 388–397. https://doi.org/10.1007/3-540-48405-1_25.
5. Brier, E.; Clavier, C.; Olivier, F. Correlation Power Analysis with a Leakage Model. In *Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2004: 6th International Workshop Cambridge, MA, USA, 11–13 August 2004*; Proceedings; Joye, M., Quisquater, J., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3156, pp. 16–29. https://doi.org/10.1007/978-3-540-28632-5_2.
6. Chari, S.; Rao, J.R.; Rohatgi, P. Template Attacks. In *Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, 13–15 August 2002*; Revised Papers; Kaliski, B.S., Koç, Ç.K., Paar, C., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2523, pp. 13–28. https://doi.org/10.1007/3-540-36400-5_3.
7. F, M.A.K.; Ganesan, V.; Bodduna, R.; Rebeiro, C. PARAM: A Microprocessor Hardened for Power Side-Channel Attack Resistance. In Proceedings of the 2020 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2020, San Jose, CA, USA, 7–11 December 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 23–34. https://doi.org/10.1109/HOST45689.2020.9300263.
8. Shaout, A.; Ahmad, O.; Al-Dulaimi, Y. AES-RV: A Low-Latency and Energy-Efficient AES Accelerator with Instruction Extension for RISC-V SoC. *arXiv* **2024**, arXiv:2505.11880.
9. Cui, S.; Balasch, J. Efficient Software Masking of AES through Instruction Set Extensions. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, 17–19 April 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 1–6. https://doi.org/10.23919/DATE56975.2023.10137150.
10. RISC-V Cryptography Extension Task Group. *RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions*; Version 0.9.3-DRAFT; RISC-V: Zurich, Switzerland, 2023.
11. Groß, H.; Mangard, S.; Korak, T. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016, Vienna, Austria, 24 October 2016*; Bilgin, B., Nikova, S., Rijmen, V., Eds.; ACM: New York, NY, USA, 2016, p. 3. https://doi.org/10.1145/2996366.2996426.
12. National Institute of Standards and Technology. Security Requirements for Cryptographic Modules. *Federal Information Processing Standards Publication*; FIPS 140-3; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2019. https://doi.org/10.6028/NIST.FIPS.140-3.
13. Lu, M.; Fan, A.; Xu, J.; Shan, W. A Compact, Lightweight and Low-Cost 8-Bit Datapath AES Circuit for IoT Applications in 28nm CMOS. In Proceedings of the 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, 1–3 August 2018; IEEE: Piscataway, NJ, USA, 2018, pp. 1464–1469. https://doi.org/10.1109/TRUSTCOM/BIGDATASE.2018.00204.
14. Dhanuskodi, S.N.; Allen, S.; Holcomb, D.E. Efficient Register Renaming Architectures for 8-bit AES Datapath at 0.55 pJ/bit in 16-nm FinFET. *IEEE Trans. Very Large Scale Integr. Syst.* **2020**, *28*, 1807–1820. https://doi.org/10.1109/TVLSI.2020.2999593.
15. Wamser, M.S.; Sigl, G. Pushing the limits further: Sub-atomic AES. In Proceedings of the 2017 IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2017, Abu Dhabi, United Arab Emirates, 23–25 October 2017; IEEE: Piscataway, NJ, USA, 2017, pp. 1–6. https://doi.org/10.1109/VLSI-SOC.2017.8203470.
16. Banik, S.; Bogdanov, A.; Regazzoni, F. Atomic-AES: A Compact Implementation of the AES Encryption/Decryption Core. *In Proceedings of the Progress in Cryptology—INDOCRYPT 2016—17th International Conference on Cryptology in India, Kolkata, India, 11–14 December 2016*; Proceedings; Dunkelman, O., Sanadhya, S.K., Eds.; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2016; Volume 10095, pp. 173–190. https://doi.org/10.1007/978-3-319-49890-4_10.
17. Moradi, A.; Poschmann, A.; Ling, S.; Paar, C.; Wang, H. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Proceedings of the Advances in Cryptology—EUROCRYPT 2011, Tallinn, Estonia, 15–19 May 2011; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6632, Lecture Notes in Computer Science, pp. 69–88.

18. Yu, J.; Aagaard, M. Benchmarking and Optimizing AES for Lightweight Cryptography on ASICs. In Proceedings of the Lightweight Cryptography Workshop, Gaithersburg, MD, USA, 4–6 November 2019.
19. Dao, M.H.; Hoang, V.P.; Dao, V.L.; Tran, X.T. An Energy Efficient AES Encryption Core for Hardware Security Implementation in IoT Systems. In Proceedings of the 2018 International Conference on ATC, Ho Chi Minh City, Vietnam, 18–20 October 2018; pp. 301–304. https://doi.org/10.1109/ATC.2018.8587500.
20. Tran, K. Integration of the AES Cryptography Extension into a RISC-V Architecture. Master's Thesis, Oklahoma State University, Stillwater, OK, USA, 2025.
21. Marshall, B.; Newell, G.R.; Page, D.; Saarinen, M.O.; Wolf, C. The design of scalar AES Instruction Set Extensions for RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2021**, *2021*, 109–136. https://doi.org/10.46586/TCHES.V2021.I1.109-136.
22. Zhang, X.; Parhi, K.K. High-speed VLSI architectures for the AES algorithm. *IEEE Trans. Very Large Scale Integr. Syst.* **2004**, *12*, 957–967.
23. Waterman, A.; Asanović, K. *The RISC-V Instruction Set Manual*; RISC-V International: Zurich, Switzerland, 2019.
24. Hojati, Z.; Jahanpeima, Z.; Rajabalipanah, M.; Ta'ati, H.; Rabiei, A.; Navabi, Z. Sharing AES Engine for RISC-V Custom Instructions Performing Encryption and Decryption. In Proceedings of the IEEE East-West Design & Test Symposium, EWDTS 2024, Yerevan, Armenia, 13–17 November 2024; IEEE: Piscataway, NJ, USA, 2024; pp. 1–6. https://doi.org/10.1109/EWDTS63723.2024.10873766.
25. Boyar, J.; Peralta, R. A small depth-16 circuit for the AES S-box. In *Proceedings of the SEC 2012*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 287–298.
26. Daemen, J.; Rijmen, V. *The Design of Rijndael: AES—The Advanced Encryption Standard*; Springer: Berlin/Heidelberg, Germany, 2002. https://doi.org/10.1007/978-3-662-04722-4.
27. Clermont, J.; Heuser, A.; Rioul, O.; Standaert, F.X. Vertical Attack Correlation: Exploiting Data Compression in Side-Channel Analysis. In *Proceedings of the IACR Transactions on Cryptographic Hardware and Embedded Systems*; Ruhr-Universität Bochum, Germany, 2021; Volume 2021, pp. 1–27. https://doi.org/10.46586/tches.v2021.i3.1-27.
28. NewAE Technology Inc. CW305 Artix FPGA Target Board. 2023. Available online: https://rtfm.newae.com/Targets/CW305%20Artix%20FPGA/ (accessed on 15 April 2023).
29. Cadence Design Systems, Inc. Cadence Genus Synthesis Solution. 2021. Available online: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html (accessed on 8 May 2021).
30. Becker, G.; Cooper, J. Test Vector Leakage Assessment (TVLA) Methodology in Practice. Available online: https://www.semanticscholar.org/paper/Test-Vector-Leakage-Assessment-(-TVLA-)-methodology-Becker-Cooper/60b993cb11fff28c9ea657b0e2882867b8f810e1 (accessed on 9 November 2023).
31. Mangard, S. Hardware Countermeasures against DPA—A Statistical Analysis of Their Effectiveness. In *Proceedings of the Topics in Cryptology—CT-RSA 2004, San Francisco, CA, USA, 23–27 February 2004*; Okamoto, T., Ed.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2004; Volume 2964, pp. 222–235. https://doi.org/10.1007/978-3-540-24660-2_18.
32. Mangard, S.; Oswald, E.; Popp, T. *Power Analysis Attacks—Revealing the Secrets of Smart Cards*; Springer: Berlin/Heidelberg, Germany, 2007.
33. Schneider, T.; Moradi, A. Leakage Assessment Methodology. In Proceedings of the Cryptographic Hardware and Embedded Systems (CHES), Saint-Malo, France, 13–16 September 2015; pp. 495–513. https://doi.org/10.1007/978-3-662-48324-4_25.
34. Trichina, E.; Seta, D.D.; Germani, L. Simplified Adaptive Multiplicative Masking for AES. In *Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, 13–15 August 2002*; Revised Papers; Kaliski, B.S., Koç, Ç.K., Paar, C., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2002; Volume 2523, pp. 187–197. https://doi.org/10.1007/3-540-36400-5_15.
35. Moradi, A.; Mischke, O.; Eisenbarth, T. Correlation-Enhanced Power Analysis Collision Attack. In *Proceedings of the Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, 17–20 August 2010*; Proceedings; Mangard, S., Standaert, F., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2010; Volume 6225, pp. 125–139. https://doi.org/10.1007/978-3-642-15031-9_9.
36. Prouff, E.; Rivain, M.; Bevan, R. Study of Second-Order Side-Channel Attacks on AES Masked Implementations. *IEEE Trans. Inf. Forensics Secur.* **2009**, *4*, 636–645. https://doi.org/10.1109/TIFS.2009.2033229.
37. Maghrebi, H. Deep Learning based Side Channel Attacks in Practice. *IACR Cryptol. ePrint Arch.* **2019**, 2019, 578.