Toward a Lightweight, Scalable, and Parallel Secure Encryption Engine

Rasha Karakchi
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, USA
karakchi@cec.sc.edu

Rye Stahle-Smith
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, USA
rye@email.sc.edu

Nishant Chinnasami
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, USA
nishantc@email.sc.edu

Tiffany Yu
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, USA
tyu@email.sc.edu

***Abstract*—The exponential growth of applications of the Internet of Things (IoT) has intensified the demand for efficient, high-throughput, and energy-efficient data processing at the edge. Conventional CPU-centric encryption methods suffer from performance bottlenecks and excessive data movement, especially in latency-sensitive and resource-constrained environments. In this paper, we present SPiME, a lightweight, scalable, and FPGA-compatible Secure Processor in Memory Encryption architecture that integrates the Advanced Encryption Standard (AES-128) directly into a Processing-in-Memory (PiM) framework. SPiME is designed as a modular array of parallel PiM units, each combining an AES core with a minimal control unit to enable distributed in-place encryption with minimal overhead.**

The architecture is fully implemented in Verilog and tested on multiple AMD UltraScale and UltraScale+ FPGAs. Evaluational results show that SPiMe can scale beyond 4,000 parallel units while maintaining less than 5% utilization of key FPGA resources on high-end devices. It delivers over 25 Gbps in sustained encryption throughput with predictable, low-latency performance. The design's portability, configurability, and resource efficiency make it a compelling solution for secure edge computing, embedded cryptographic systems, and customizable hardware accelerators.

***Index Terms*—FPGA, Verilog, AES, Processor-in-Memory**

I. Introduction

The rise of Internet of Things (IoT) systems has significantly accelerated Big Data generation, raising urgent challenges in secure, real-time, and energy-efficient data processing [1]. IoT devices continuously produce sensitive data streams that must be encrypted and processed under tight bandwidth and energy constraints, exposing the limitations of conventional CPU-centric systems in terms of latency and data movement overhead.

Processing-in-Memory (PiM) has emerged as a promising alternative, enabling computation near data to reduce transfer overhead and improve performance [1]. While PiM has demonstrated benefits in domains such as pattern matching [2]–[5], genomics [6], and AI inference [7], securing data within PiM remains a critical challenge. Traditional memory hierarchies and centralized encryption methods fail to meet the urgent needs of modern IoT and edge computing environments [8], [9].

Security in PiM systems is particularly sensitive due to the risk of data exposure during transfer, susceptibility to side-channel attacks, and power leakage [10]. The Advanced Encryption Standard (AES) is a widely adopted countermeasure, but software-based AES is computationally intensive and ill-suited for real-time applications [11]. Embedding AES directly into PiM architectures mitigates these issues by protecting data in place, reducing latency, and minimizing the observable attack surface [12], [13].

In this work, we introduce **SPiME**, a scalable and lightweight AES-128 encryption system embedded within a PiM framework and implemented in Verilog. Each PiM unit integrates an `aes_core`—comprising submodules `sub_bytes`, `shift_rows`, and `mix_columns`—with a `pim_controller` for key scheduling and I/O. The system supports throughput scaling via a parameterized `NUM_PIMs` variable, enabling parallel encryption across distributed memory banks. SPiMe is designed for FPGA compatibility, modular reuse, and energy-efficient secure computation at the edge. It addresses critical performance and security requirements for PiM-based IoT platforms, providing a practical foundation for secure, high-throughput data processing.

II. Related Work

AES has been widely implemented in hardware to improve performance, energy efficiency, and area utilization [14]–[27]. Prior designs have focused on optimizing resource-constrained implementations [11], [28], improving resistance to side-channel attacks through balanced logic [13], and achieving high throughput using pipelined and parallel structures [8], [29]. Other efforts are application-specific, such as targeting 5G networks [12] or employing approximate computing to

save energy [10], though these may compromise security. Xu et al. [9] introduced a PiM-based AES integrated into DRAM, but their approach relies on custom memory technology, limiting portability and compatibility with FPGA platforms.

Chaves et al. [30] presented a polymorphic AES core that merges SubBytes and MixColumns operations to reduce resource usage and improve throughput. Iranfar et al. [31] developed a spintronic-based AES design for PiM, which offers strong resistance to power-based side-channel attacks through a symmetrical logic structure and uniform power profile. Liu et al. [32] proposed AESPIM for real-time video encryption, incorporating system-level enhancements such as data/user-level parallelism and QoS-aware scheduling to improve streaming performance. Instruction-level approaches [33] leverage AES-specific ISA extensions or general-purpose operations like Pread and byte_perm, achieving fast encryption on CPUs but relying on specialized hardware and typically supporting only non-feedback encryption modes.

In contrast, our work introduces a modular and reconfigurable FPGA-based AES architecture designed for Processing-in-Memory (PiM) systems. By integrating multiple AES-128 cores with lightweight controllers, our design enables parallel and pipelined encryption directly near memory. This significantly reduces data movement and memory access overhead, leading to improved throughput and energy efficiency. Unlike solutions that require custom memory or CPU instruction extensions, our architecture supports flexible deployment across FPGA platforms, making it well-suited for secure, scalable, and energy-aware applications in edge and embedded environments.

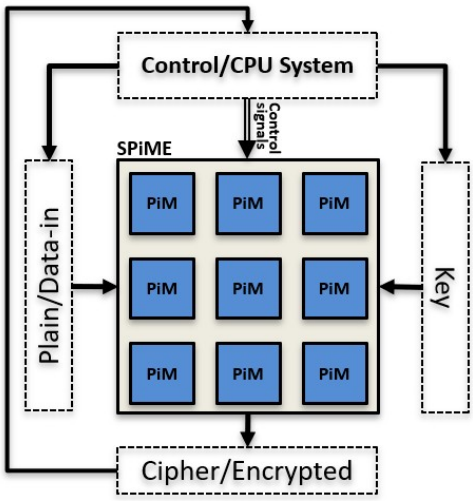

Fig. 1: Proposed SPiME System.

III. Design and Architectures

Our proposed architecture SPiME (Secure Processor in Memory Encryption) is designed as an array of parallel multiple processor-in-memory (PiM) units where each PiM comprises a PiM controller and an AES core. The design is fully described in Verilog and features a modular organization that promotes scalability, reusability, and clarity.

Figure 1 illustrates the full SPiME architecture, composed of an array of multiple PiM units. Each PiM unit receives its plaintext input (data_in) and encryption key from buffers managed by the CPU. The operation of the PiM units is directed by a set of control signals originating from the CPU. After processing, each PiM unit outputs the ciphertext to a corresponding output buffer. The experimental evaluation presented in this work focuses specifically on the SPiME component itself, excluding analysis of the resource usage or performance of the surrounding system. The followning subsections are description of SPiME and its components.

A. Top-Level System

This is the main module that instantiates multiple PiM processing units, each consisting of a PiM_Controller and an associated AES_Core. It includes a parameter, NUM_PiMs, which defines the number of parallel encryption units operating in the system. The top-level module coordinates global input/control signals such as system clock, reset, encryption start, and plaintext/key input. It gathers outputs from each processing unit and can be extended to include aggregation, output buffering, or interfacing to a larger memory subsystem. Figure 2 presents the design units of n processor-in-memory units.

B. PiM Controller

Each PiM_Controller is responsible for orchestrating the AES encryption process for its assigned memory block. As shown in Algorithm 1, it handles:

Start/Done Handshaking initiates the AES core when start is asserted, and monitors the done signal from the AES core.

Data Handling routes the input plaintext and key to the AES core and captures the result of the ciphertext after encryption.

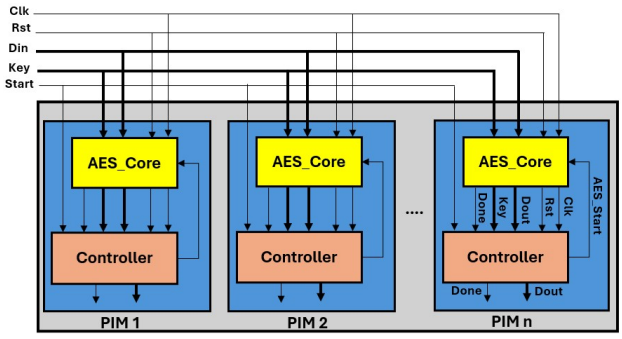

Fig. 2: SPiME top_Level design consisting of an *n* PiMs where each consists of AES_Core and control unit

Control Flow Management can support additional logic for data buffering, memory interfacing, or result distribution in a complete PIM setup. The controller acts as a lightweight scheduler that isolates control complexity from the AES datapath and makes the design modular.

Algorithm 1: PIM Controller State Machine

```
Function PIM_Controller(clk, rst, start, data_in, key, aes_done,
  aes_data_out):
    // Initialize state
    state ← IDLE;
    aes_start ← 0;
    done ← 0;
    data_out ← 0;
    while true do
        if rst = 1 then
            state ← IDLE;
            aes_start ← 0;
            done ← 0;
        else
            switch state do
                case IDLE do
                    if start = 1 then
                        aes_start ← 1;
                        state ← START_AES;
                case START_AES do
                    aes_start ← 0;
                    state ← WAIT_AES;
                case WAIT_AES do
                    if aes_done = 1 then
                        data_out ← aes_data_out;
                        done ← 1;
                        state ← DONE;
                case DONE do
                    done ← 0;
                    state ← IDLE;
```

C. AES_Core

It performs AES-128 encryption on a single 128-bit data block using a 128-bit key. It is pipelined and driven by a finite-state machine (FSM) that sequences through the AES rounds. In this design, we depend on the previous work of AES [14]–[27]. The `AES_Core` is designed as a synchronous digital hardware module implemented in Verilog that controls the AES encryption process. This module coordinates the sequence of AES transformations applied to the input plaintext using a set of pre-computed round keys.

- **Inputs:**
  - `clk`: System clock signal drives the sequential logic.
  - `rst`: Active-high synchronous reset signal to initialize internal states.
  - `start`: Signal to initiate the encryption process.
  - `data_in`: 128-bit plaintext input data.
  - `key`: 128-bit AES key (used for key expansion).
  - `round_keys_flat`: Concatenated 1408-bit bus containing all 11 round keys (each 128-bit).
- **Outputs:**
  - `data_out`: 128-bit ciphertext output after encryption.
  - `done`: Flag indicating completion of encryption.

Upon initialization or reset, the module unpacks the `round_keys_flat` input into an array `round_keys[0..10]`, each holding a 128-bit round key. The module includes a finite-state machine (FSM) with four states: `IDLE`, `INIT`, `ROUND`, and `FINAL`, which control the encryption flow:

IDLE State: The module waits for the `start` signal. When asserted, it transitions to the `INIT` state. Outputs remain inactive during this state.

INIT State: Resets the round counter to zero and performs the initial AddRoundKey operation by XOR-ing the input data with `round_keys[0]`, initializing the internal AES state. Then, it moves to the `ROUND` state.

ROUND State: Sequentially performs AES round transformations:*SubBytes*, *ShiftRows* and *MixColumns*. The transformed state is XOR-ed with the next round key `round_keys[round + 1]`, and the round counter is incremented. If the round counter reaches 9, the FSM transitions to the `FINAL` state; otherwise, it continues processing rounds.

FINAL State: Executes the final AES round, applying *SubBytes* and *ShiftRows* but skipping *MixColumns*. The state is then XOR-ed with the last round key `round_keys[10]`. The output `data_out` is set to the encrypted data, `done` is asserted, and the FSM returns to `IDLE`.

When the synchronous reset `rst` is asserted, all internal registers, including the state machine and round counter, are cleared. The FSM returns to the `IDLE` state, and the `done` signal is de-asserted. The AES_Core is divided into the following submodules that represent the functional blocks of AES. Each submodule is designed to be synthesized and modular. Algorithm 2 describes thoroughly the AES_Core operation.

D. AES_Sub_Bytes

This module implements the AES S-Box substitution operation. Each byte in the 128-bit block is replaced using a lookup table or logic circuit that performs the non-linear transformation. The S-Box is designed using combinational logic or ROM-based lookup depending on FPGA resources. The `AES_Sub_Bytes` module implements the SubBytes transformation step in AES encryption as a synchronous process controlled by a clock and reset signals. It handles input data packets, validates them, and produces corresponding outputs based on the packet type.

- **Inputs:**
  - `clock`: The system clock signal, driving all sequential operations.
  - `reset`: Active-high synchronous reset signal to initialize internal registers.
  - `input_valid`: A flag indicating when the input data is valid.
  - `packet_type`: A signal identifying the type of input packet.
  - `input_data`: The data input to be processed by the SubBytes operation.
- **Outputs:**
  - `output_valid`: Flag indicating when the output data is valid.
  - `output_data`: The processed output data after the SubBytes step.

Algorithm 2: AES Core Algorithm

Function AES_Core($clk, rst, start, data\_in, key, round\_keys\_flat$):
  Input: clk, rst, $start$, $data\_in$, key, $round\_keys\_flat$
  Output: $done$, $data\_out$
  // Unpack round keys from flattened input
  for $i \leftarrow 0$ **to** 10 **do**
    $round\_keys[i] \leftarrow round\_keys\_flat[i \cdot 128+ : 128]$;
  // Initialize FSM
  $current\_state \leftarrow IDLE$;
  // FSM transition
  if rst **then**
    $current\_state \leftarrow IDLE$;
  else
    $current\_state \leftarrow next\_state$;
  // FSM next state logic
  if $current\_state = IDLE$ **then**
    $next\_state \leftarrow start\ ?\ INIT : IDLE$;
  else if $current\_state = INIT$ **then**
    $next\_state \leftarrow ROUND$;
  else if $current\_state = ROUND$ **then**
    $next\_state \leftarrow (round = 9)\ ?\ FINAL : ROUND$;
  else if $current\_state = FINAL$ **then**
    $next\_state \leftarrow IDLE$;
  else
    $next\_state \leftarrow IDLE$;
  // Round logic
  if rst **then**
    $state \leftarrow 0$;
    $round \leftarrow 0$;
    $done \leftarrow 0$;
  else
    if $current\_state = IDLE$ **then**
      $done \leftarrow 0$;
    else if $current\_state = INIT$ **then**
      $round \leftarrow 0$;
      $state \leftarrow data\_in \oplus round\_keys[0]$;
    else if $current\_state = ROUND$ **then**
      $state \leftarrow mix\_columns\_out \oplus round\_keys[round+1]$;
      $round \leftarrow round + 1$;
    else if $current\_state = FINAL$ **then**
      $state \leftarrow shift\_rows\_out \oplus round\_keys[10]$;
      $data\_out \leftarrow state$;
      $done \leftarrow 1$;
  // Apply AES transformations
  $sub\_bytes\_out \leftarrow SubBytes(state)$;
  $shift\_rows\_out \leftarrow ShiftRows(sub\_bytes\_out)$;
  $mix\_columns\_out \leftarrow MixColumns(shift\_rows\_out)$;
  return $done$, $data\_out$

The module operates as follows:

- When the `reset` signal is asserted (high), the module clears its internal temporary data register and de-asserts the `output_valid` signal to zero, effectively resetting its internal state.
- On each rising edge of the `clock` signal, if the `input_valid` flag is high, the module inspects the `packet_type`:
  - If the `packet_type` equals 2 (indicating the packet is relevant for the SubBytes operation), the input data is latched into an internal temporary register, and the `output_valid` flag is asserted to indicate the output data is now valid.
  - Otherwise, if the packet type does not match, `output_valid` is de-asserted, indicating no valid output data for this cycle.
- The output data, `output_data`, continuously reflects the value stored in the temporary register `temp_data`.

This behavior ensures that only valid data packets of the expected type are processed and output, while other packets are ignored.

E. AES_Shift_Rows

This module implements the cyclic left shift of the AES state rows. The `AES ShiftRows` transformation is a permutation step applied to the 128-bit AES state matrix. It cyclically shifts the bytes in each row of the state by a certain offset to the left, depending on the row index.

- **Input:** A 128-bit state `data_in` represented as a 4x4 matrix of bytes.
- **Output:** A 128-bit state `data_out`, also represented as a 4x4 matrix of bytes after applying the ShiftRows operation.

The transformation proceeds as follows:

- The state matrix consists of 4 rows and 4 columns, where each element is a byte.
- For the first row (`row = 0`), the bytes remain unchanged; `data_out[0][column]` is directly assigned from `data_in[0][column]`.
- For subsequent rows (`row = 1, 2, 3`), each byte is shifted cyclically to the left by an amount equal to the row index:

$$\texttt{data\_out[row][column]} \leftarrow \texttt{data\_in[row]}[(column + row) \bmod 4]$$

  meaning each row shifts its bytes left by its row number, wrapping around cyclically.

F. AES_Mix_Columns

The AES_ Mix_Columns transformation operates on the 128-bit input state `data_in`, which is arranged as 4 columns of 4 bytes each. The transformation processes each column independently by applying finite field arithmetic in GF(2^8).

Specifically, for each column c (from 0 to 3), the four bytes (s_0, s_1, s_2, s_3) of that column are extracted. Each output byte (m_0, m_1, m_2, m_3) of the transformed column is computed as a linear combination of the input bytes using multiplication by constants 2 and 3 in GF(2^8), where multiplication by 2 is implemented by the function `mul_by_2` and multiplication by 3 is performed by `mul_by_3`. These computations are defined as follows:

$$m_0 = \text{mul\_by\_2}(s_0) \oplus \text{mul\_by\_3}(s_1) \oplus s_2 \oplus s_3$$
$$m_1 = s_0 \oplus \text{mul\_by\_2}(s_1) \oplus \text{mul\_by\_3}(s_2) \oplus s_3$$
$$m_2 = s_0 \oplus s_1 \oplus \text{mul\_by\_2}(s_2) \oplus \text{mul\_by\_3}(s_3)$$
$$m_3 = \text{mul\_by\_3}(s_0) \oplus s_1 \oplus s_2 \oplus \text{mul\_by\_2}(s_3)$$

Here, multiplication by 2 (mul_by_2) is implemented as a left shift of the byte followed by a conditional XOR with 0x1b if the most significant bit was set before the shift, to ensure reduction modulo of the AES polynomial. Multiplication by 3 (`mul_by_3`) is computed as the XOR of `mul_by_2` and the original byte. After computing (m_0, m_1, m_2, m_3), the output

state `data_out` is updated by replacing column c with these new bytes. This process is repeated for all four columns, resulting in the fully transformed AES state.

G. Add_Round_Key and Key Scheduler

It performs XOR between the 128-bit state and the 128-bit round key. This operation is simple but crucial for combining the input data with the key material. Although not always implemented as a separate module, the AES_Core contains logic to expand the 128-bit key into 11 round keys using the Rijndael key expansion algorithm [34]. Each round key is used once per round.

IV. Testing and Evaluation Process

To assess the practicality and performance of the proposed SPiMe architecture, we performed a detailed evaluation using multiple FPGA platforms. Our analysis spans hardware resource utilization, scalability, latency, and throughput, with designs synthesized and tested on AMD UltraScale and UltraScale+ devices.

A. FPGA Platforms and Configuration

We selected five FPGA platforms for evaluation: U55C, U280, VCU118, ZCU104, and ZCU106. These devices span a range from high-end data center accelerators to embedded-class SoCs. Table I summarizes their hardware specifications, including available logic (LUTs), flip-flops (FFs), memory resources (BRAM and URAM), and DSP blocks.

To explore SPiME's scalability, we instantiated arrays with increasing numbers of parallel PiM units: 256, 512, 1024, 2048, and 4096. Each PIM unit comprises an AES-128 encryption core and a control unit (controller). The upper bound of 4096 units was selected based on routing limitations observed during placement and implementation, particularly due to the total wire count exceeding 1 million nets in the largest configurations.

TABLE I: Comparison of FPGA Devices

Device	Part	LUTs (K)	FFs (K)	BRAM	URAM	DSPs
U55C	xcu55c-fsvh2892-2L-e	1304	2607	2016	960	9024
U280	xcu280-fsvh2892-2L-e	1304	2607	2016	960	9024
VCU118	xcvu9p-flga2104-2L-e	1182	2364	2160	960	6840
ZCU104	xczu7ev-ffvc1156-2-e	230	460	312	96	1728
ZCU106	xczu7ev-ffvc1156-2-e	230	460	312	96	1728

B. Hardware Resource Utilization

Figure 3 illustrates the LUT utilization across different FPGA platforms as the number of PIM units increases. On high-end devices like U55C, U280, and VCU118, even the 4096-PIM configuration consumed only around 3.65% of the available LUTs. In contrast, smaller devices such as ZCU104 and ZCU106 reported LUT utilization near 18.44% for the same configuration, highlighting the impact of limited logic capacity.

Figure 4 presents the flip-flop (register) usage. As with LUTs, register consumption scales linearly with the number of instantiated PiM units. Larger FPGAs maintained sub-2% FF utilization for 4096 PiMs, while ZCU-series devices reached nearly 10%. These results confirm the modularity and efficiency of the SPiME design, indicating that the per-PiM overhead in both logic and control resources remains consistent across platforms.

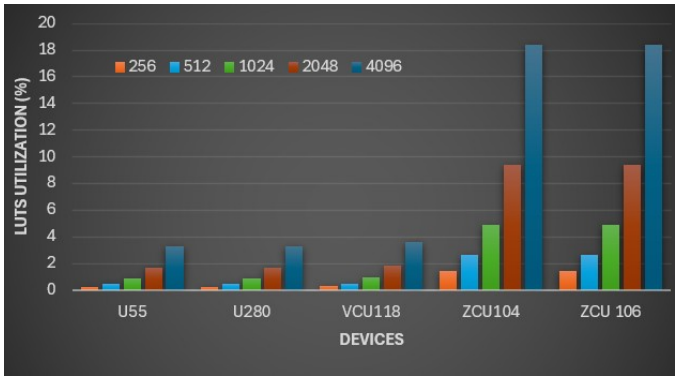

Fig. 3: The LUTs utilization percentage versus number of PiMs (256-4096) on different FPGA devices.

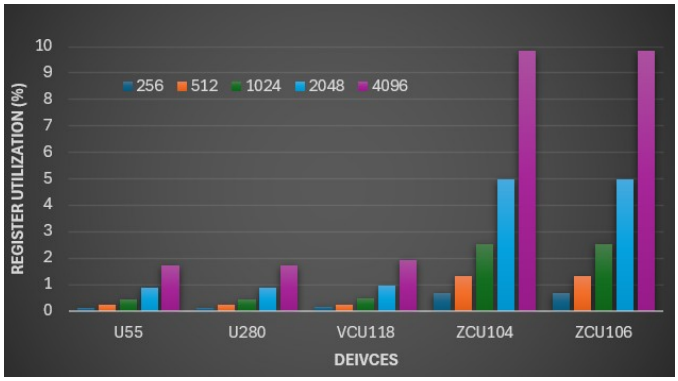

Fig. 4: The register utilization percentage versus number of PiMs (256-4096) on different FPGA devices.

C. Latency Analysis

Latency was calculated as a function of the number of cycles per AES operation and the operating frequency (Fmax). Each AES operation requires a fixed 11-cycle sequence: 1 cycle for initiation, 9 for AES rounds, and 1 for finalization. The latency in microseconds is given by Equation 1:

$$\text{Latency } (\mu s) = 1000 \times \left(\frac{\text{Cycles to execute one task}}{f_{\max}} \right) \quad (1)$$

Figure 5 shows how latency inversely scales with clock frequency. At 100 MHz, latency is approximately 0.11 μ s, reducing to 0.036 μ s and 0.022 μ s at 300 MHz and 500 MHz, respectively. This behavior confirms SPiMe's suitability for real-time applications, as its latency remains both low and predictable.

D. Throughput Evaluation

Throughput was measured as the amount of encrypted data (in bits) divided by latency. The throughput in Gbps is given by Equation 2:

$$\text{Throughput (Gbps)} = \left(\frac{\text{Block size}}{\text{Latency } (\mu s)} \right) \div 10^6 \qquad (2)$$

Figure 6 shows the performance as the number of PIM units increases, using a fixed block size of 1024 bits. Throughput scales nearly linearly with PIM count, reaching over 23 Gbps at 500 MHz with 4096 PIMs.

Figure 7 shows the effect of increasing the block sizes (1K, 4K, 16K, 64K) at a fixed frequency and PIM count. Larger blocks amortize control and I/O overhead, yielding significantly higher throughput. The design performs best in batch or buffered processing scenarios, which are common in secure IoT edge devices.

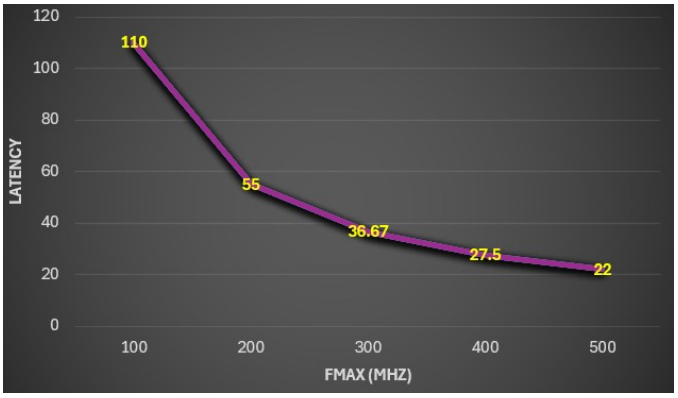

Fig. 5: SPiME latency varies based on the Fmax while same for all arrays of NUM_PIMs.

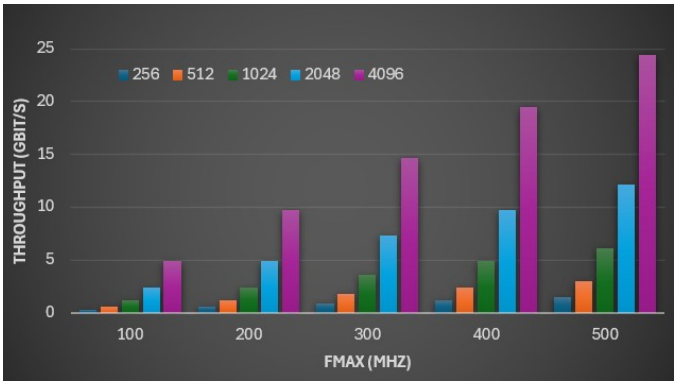

Fig. 6: Throughput of 1024 block size with the NUM_PIMs varies from 1K to 4K.

The results show that SPiMe efficiently scales up to 4096 PIMs with minimal LUT and FF overhead. The latency is deterministic and low, ranging from 0.022 μ s to 0.11 μ s depending on frequency. Throughput exceeds 25 Gbps in optimal configurations, with consistent linear scaling. The

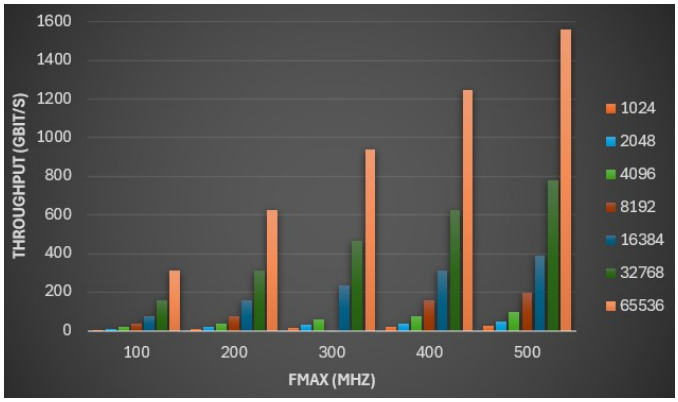

Fig. 7: Throughput varies based on the block size while maximum frequency and num_pims varies.

architecture remains portable across large and small FPGA platforms with proportional resource usage.

V. Conclusion and Future Work

This work presented SPiME, a scalable, lightweight AES-based secure memory core tailored for FPGA-based Processor-in-Memory (PiM) architectures. Implemented entirely in Verilog, SPiME integrates a PIM controller with an AES-128 core, enabling parallel encryption with minimal control overhead.

Hardware Efficiency and Scalability: SPiME exhibits excellent scalability across a range of FPGA platforms. On high-end devices like the U55C and VCU118, we instantiated up to 4096 parallel PiM units with under 4% resource utilization. On smaller platforms (ZCU104/106), SPiME scaled down effectively, maintaining low per-unit overhead. The design's modularity ensures adaptability from embedded systems to datacenter-class accelerators.

Latency and Throughput: Each AES encryption completes in a fixed 11-cycle sequence, resulting in predictable, constant-time performance. At 500 MHz, latency drops to 0.022 µs. Throughput scales linearly with the number of units and block size, reaching over 25 Gbps at peak with 4K units. Larger blocks significantly improve throughput efficiency by amortizing control overhead, making SPiME suitable for secure streaming and batched workloads like video analytics and edge AI inference.

Robustness: No architectural bottlenecks were observed across scale tests. FSM-based control and pipelining enabled smooth operation, with routing congestion being the only limitation at extreme scales—an issue addressable in ASIC flows or future FPGAs with improved routing fabrics.

Future work includes full system integration with memory and CPU coordination, dynamic workload adaptation, real hardware benchmarking, and side-channel security validation. Additionally, SPiME will be extended with high-level software APIs to support broader adoption in edge and cloud-based secure computing environments.

To conclude, SPiME is, to our knowledge, the first FPGA-compatible, parameterizable PiM-based encryption core that

supports scalable parallel AES processing. Its modular design, predictable performance, and low overhead make it a strong candidate for secure, high-throughput processing in both edge and cloud environments.

REFERENCES

[1] Xu Yang, Yumin Hou, and Hu He. A processing-in-memory architecture programming paradigm for wireless internet-of-things applications. *Sensors*, 19(1):140, 2019.

[2] Rasha Karakchi and Jason D. Bakos. Napoly: A non-deterministic automata processor overlay. *ACM Transactions on Reconfigurable Technology and Systems*, 16:1–25, 2023.

[3] Rasha Karakchi, Lothrop O. Richards, and Jason D. Bakos. A dynamically reconfigurable automata processor overlay. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, 2017.

[4] Rasha Karakchi, Charles Daniels, and Jason Bakos. An overlay architecture for pattern matching. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, volume 2160-052X, pages 165–172, 2019.

[5] Ryan Karbowniczak and Rasha Karakchi. Optimizing sequence alignment with scored nfas. *arXiv preprint arXiv:2501.02162*, 2025.

[6] Rasha Karakchi, Jordan A. Bradshaw, and Jason D. Bakos. High-level synthesis of a genomic database search engine. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2016.

[7] Rasha Karakchi and Ryan Karbowniczak. Developing a self-explanatory transformer. In *2024 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 523–525. IEEE, 2024.

[8] Kimmo Järvinen, Matti Tommiska, and Jouni Skyttä. A fully pipelined memoryless 17.8 gbps aes-128 encryptor. In *Field Programmable Logic and Applications (FPL)*, pages 147–152. IEEE, 2008.

[9] Lei Xu, Hao Wang, and Yong Chen. A processing-in-memory aes implementation in dram for secure and efficient data encryption. *IEEE Transactions on Computers*, 2023. Early Access.

[10] Jun Zhang, Yu Liu, and Jie Han. An energy-efficient aes implementation using approximate logic synthesis. *Integration, the VLSI Journal*, 75:85–94, 2021.

[11] Robert McEvoy, Conor Murphy, Máire McLoone, and William Marnane. A compact fpga-based architecture for aes encryption. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(7):693–701, 2006.

[12] Qiang Wang, Li Zhang, and Yifan Zhao. High-performance aes-gcm design for 5g security on fpga. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–18, 2021.

[13] Kris Tiri and Ingrid Verbauwhede. Securing encryption algorithms against dpa at the logic level: Next generation smart card technology. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2003.

[14] Paweł Chodowiec and Kris Gaj. Very compact fpga implementation of the aes algorithm. In *International workshop on cryptographic hardware and embedded systems*, pages 319–333. Springer, 2003.

[15] Ashwini M Deshpande, Mangesh S Deshpande, and Devendra N Kayatanavar. Fpga implementation of aes encryption and decryption. In *2009 international conference on control, automation, communication and energy conservation*, pages 1–6. IEEE, 2009.

[16] S Sridevi Sathya Priya, P Karthigaikumar, and Narayana Ravi Teja. Fpga implementation of aes algorithm for high speed applications. *Analog integrated circuits and signal processing*, pages 1–11, 2022.

[17] Joseph Zambreno, David Nguyen, and Alok Choudhary. Exploring area/delay tradeoffs in an aes fpga implementation. In *International Conference on Field Programmable Logic and Applications*, pages 575–585. Springer, 2004.

[18] Piotr Chodowiec and Krzysztof Gaj. Asic implementation of the aes rijndael algorithm. In *International Conference on Field Programmable Logic and Applications*, pages 160–171. Springer, 2002.

[19] Tim Good and Mohammed Benaissa. Aes on fpga from the fastest to the smallest. In *Cryptographic Hardware and Embedded Systems–CHES 2005: 7th International Workshop, Edinburgh, UK, August 29–September 1, 2005. Proceedings 7*, pages 427–440. Springer, 2005.

[20] Harshali Zodpe and Ashok Sapkal. An efficient aes implementation using fpga with enhanced security features. *Journal of King Saud University-Engineering Sciences*, 32(2):115–122, 2020.

[21] Taniya Hasija, Amanpreet Kaur, KR Ramkumar, Shagun Sharma, Sudesh Mittal, and Bhupendra Singh. A survey on performance analysis of different architectures of aes algorithm on fpga. *Modern Electronics Devices and Communication Systems: Select Proceedings of MEDCOM 2021*, pages 39–54, 2023.

[22] Hrushikesh S Deshpande, Kailash J Karande, and Altaaf O Mulani. Efficient implementation of aes algorithm on fpga. In *2014 International Conference on Communication and Signal Processing*, pages 1895–1899. IEEE, 2014.

[23] Atul M Borkar, RV Kshirsagar, and MV Vyawahare. Fpga implementation of aes algorithm. In *2011 3rd International Conference on Electronics Computer Technology*, volume 3, pages 401–405. IEEE, 2011.

[24] Umer Farooq and M Faisal Aslam. Comparative analysis of different aes implementation techniques for efficient resource usage and better performance of an fpga. *Journal of King Saud University-Computer and Information Sciences*, 29(3):295–302, 2017.

[25] Xiwei Zhang, Meng Li, and Jing Hu. Optimization and implementation of aes algorithm based on fpga. In *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, pages 2704–2709. IEEE, 2018.

[26] Hrushikesh S Deshpande, Kailash J Karande, and Altaaf O Mulani. Area optimized implementation of aes algorithm on fpga. In *2015 International Conference on Communications and Signal Processing (ICCSP)*, pages 0010–0014. IEEE, 2015.

[27] Joseph Sunil, HS Suhas, BK Sumanth, and S Santhameena. Implementation of aes algorithm on fpga and on software. In *2020 IEEE International Conference for Innovation in Technology (INOCON)*, pages 1–4. IEEE, 2020.

[28] Atsushi Satoh, Shuji Morioka, Kohji Takano, and Sumio Munetoh. A compact Rijndael hardware architecture with S-box optimization. In *Advances in Cryptology—ASIACRYPT 2001*, pages 239–254. Springer, 2001.

[29] Xiaohui He, Bin Li, and Yong Zhang. A parallel aes architecture for high-speed network security. *IEEE Access*, 8:21725–21735, 2020.

[30] Ricardo Chaves, Georgi Kuzmanov, Stamatis Vassiliadis, and Leonel Sousa. Reconfigurable memory based aes co-processor. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 8–pp. IEEE, 2006.

[31] Pegah Iranfar, Abdolah Amirany, and Mohammad Hossein Moaiyeri. Power attack-immune spintronic-based aes hardware accelerator for secure and high-performance pim architectures. *IEEE Transactions on Magnetics*, 61(4):1–12, 2025.

[32] Yiding Liu, Guangyu Huang, Yuwei Zhang, Xuehai Wang, and Yu Wang. Enabling PIM-based AES encryption for online video streaming. *Journal of Systems Architecture*, 132:102734, 2022.

[33] Ruby B Lee and Yu-Yuan Chen. Processor accelerator for aes. In *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pages 16–21. IEEE, 2010.

[34] Tariq Jamil. The rijndael algorithm. *IEEE potentials*, 23(2):36–38, 2004.