Sealer: In-SRAM AES for High-Performance and Low-Overhead Memory Encryption

Jingyao Zhang
Department of Computer Science
University of California, Riverside
jzhan502@ucr.edu

Hoda Naghibijouybari
Department of Computer Science
Binghamton University
hnaghibi@binghamton.edu

Elaheh Sadredini
Department of Computer Science
University of California, Riverside
elaheh@cs.ucr.edu

***Abstract*—To provide data and code confidentiality and reduce the risk of information leak from memory or memory bus, computing systems are enhanced with encryption and decryption engine. Despite massive efforts in designing hardware enhancements for data and code protection, existing solutions incur significant performance overhead as the encryption/decryption is on the critical path. In this paper, we present *Sealer*, a high-performance and low-overhead in-SRAM memory encryption engine by exploiting the massive parallelism and bitline computational capability of SRAM subarrays. *Sealer* encrypts data before sending it off-chip and decrypts it upon receiving the memory blocks, thus, providing data confidentiality. Our proposed solution requires only minimal modifications to the existing SRAM peripheral circuitry. *Sealer* can achieve up to two orders of magnitude throughput-per-area improvement while consuming 3× less energy compared to prior solutions.**

I. Introduction

Healthcare organizations, businesses, and governments rely heavily on secure computer systems for their daily activities and business conduct. To provide data confidentiality, a wide range of computational devices, from high-end servers to low-power IoT devices, implement data encryption standards. In these devices, only the processor chip is considered secure and trusted hardware in the system [1]. Anything outside the processor chip boundary is typically assumed vulnerable and untrusted. In such a threat model, any data sent off-chip (i.e., to the memory or on the cloud) is potentially at risk of being manipulated, tampered with, or leaked [2]. Novel memory technologies, such as Non-Volatile Memories (NVMs), can retain data for a long time after power loss, thus, making memory protection even more critical.

To protect against memory and bus attacks (e.g., bus snooping and cold boot attacks [2]), processor vendors are increasingly adding support for protecting the integrity and confidentiality of data through integrity verification and encryption on the on-chip memory controller [3], [4]. Memory encryption typically uses Advanced Encryption Standard (AES) block cipher to encrypt plaintext going to or decrypt ciphertext from the main memory, using a key re-generated on each system reboot. However, real-time encryption or decryption at every memory access in the critical path results in severe performance overhead [5].

To address the performance issue, recent prior works have proposed in-memory and near-memory encryption/decryption solutions [6]–[9]. Xie et al. [6] propose AIM, an in-memory AES engine that provides bulk encryption of data blocks in NVM for mobile devices, and encryption is executed only when the device is shut down or put into sleep/screen-lock mode. Despite its computational efficiency, AIM does not protect data confidentiality against bus and memory attacks. Aga et al. [7] present InvisiMem, which uses the logic layer in 3D stacked memory to implement cryptographic primitives. However, the InvisiMem design expands the trusted computing base (TCB) to the logic layer of the memory. To protect data confidentiality against physical attacks on both memory and bus, an efficient real-time encryption/decryption engine needs to be implemented on the processor chip.

To provide a low-cost, and real-time on-chip encryption engine, for the first time, this paper proposes to re-purpose 6T SRAM subarrays into active large vector computational units to perform encryption and decryption on-chip. Our solution, *Sealer*, exploits intrinsic parallelism and bitline computational capability of memory subarrays for fast and low-overhead AES implementation, and incurs only a negligible area overhead (less than 1.55%) compared to the traditional SRAM arrays. Moreover, *Sealer* provides the same level of protection for memory confidentiality as Intel Memory Encryption Engine (MEE) [3] or AMD Secure Memory Encryption (SME) [4], and unlike InvisiMem, does not extend the TCB.

In summary, the paper makes the following **contributions:**

- We present *Sealer*, a real-time in-SRAM AES engine to provide data confidentiality by encrypting the plaintext on the CPU chip. Our proposed architecture effectively stores the required data for encryption into the same subarray. This allows the peripherals and resources to be shared among different computation and communication units, thus, reducing performance and hardware overhead.
- We present an algorithm and architecture methodology to efficiently map the AES algorithm to the *Sealer* architecture. By fusing the *SubBytes* and *ShiftRows* stages in AES, we can interleave the computation at a finer granularity and maximally exploit on-the-fly computation to significantly reduce data movement and *write* cycles.
- We compare *Sealer* with several on-chip and in-memory AES encryption engines and show that our solution has up to 323× performance improvement, up to 91× throughput-per-area improvement, and 3× lower energy

consumption compared to prior solutions. To separate the architectural and technology contribution of *Sealer*, we evaluate an in-NVM AES engine on SRAM (*AIM-SRAM*) and find that our solution achieves 6× higher performance than AIM-SRAM due to architectural contribution and 18× better performance due to technology benefits.

II. Background and Threat Model

A. Advanced Encryption Standard

The Advanced Encryption Standard (AES) in cryptography, also known as Rijndael cryptography, is a block encryption standard. The AES encryption process operates on a 4×4 matrix of bytes whose initial value is a plaintext block (the element size in the matrix is one byte). Each AES encryption round (except the last one) consists of four steps where the output of each stage is used as an input of the next stage and described as follows. (1) *AddRoundKey*: each byte in the matrix is XORed with the round key. Each round key is generated by the key generation scheme from a given cipher key. (2) *SubBytes*: each byte is substituted with its corresponding byte in the S-box block using a nonlinear substitution function commonly implemented with lookup tables (LUTs). (3) *ShiftRows*: a round-robin shift is performed for each row in the matrix. The first row is unchanged while each element of the second row are shifted to the left by 1 byte. Then for the third and fourth row, elements are shifted to the left by 2 and 3 bytes, respectively (see matrix D1 to D2 transformation in Figure 1). (4) *Mixcolumns*: this step uses a linear transformation to fully mix the four bytes of each column. In AES, the *MixColumns* stage is omitted from the last encryption loop and replaced with another *AddRoundKey*.

The overall process of AES for a 128-bit plaintext is shown in Figure 1. First, the plaintext is XORed with the original cipher key for initialization. Then after 9 main rounds, each of which consists of *AddRoundKey*, *SubBytes*, *ShiftRows*, and *MixColumns*, and the final round without *MixColumns*, the 128-bit ciphertext is generated.

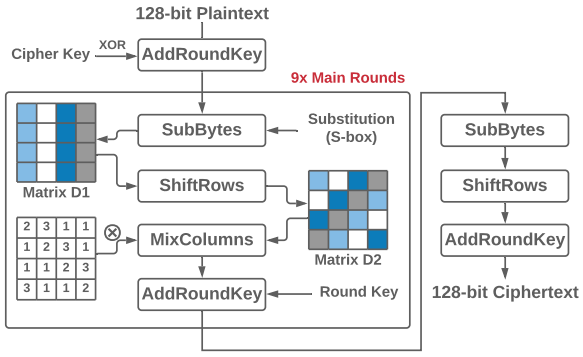

Fig. 1: The steps of AES for a 128-bit plaintext.

B. Computing in SRAM

In-SRAM computing relies on the underlying bitline computation by activating more than one row in the SRAM subarray [10]. The bitwise AND and NOR operations in SRAM are implemented directly by utilizing sense amplifiers (SAs) with multiple wordlines activated, as shown in Figure 2(a). The SA on the bitline (BL) can sense a voltage higher than V_{ref} only if all the cells in the activated rows connected to the corresponding BL contain '1'. This means the SA will sense '1', thus, achieving element-wise AND operation. The SA on the bitline-bar ($\overline{BL}$) will sense a voltage higher than V_{ref} only if all the cells in the activated rows connected to the corresponding $\overline{BL}$ contain '1', which, in turn, implies that all the cells in the activated rows connected to the corresponding BL contain '0'. This means the SA will sense '1', thus, achieving element-wise NOR operation. Using the logical bitwise AND and NOR operations, the XOR operation can be performed, as shown in Figure 2(b).

Many studies based on computing in SRAM have been proposed [11]–[14]. Cache Automaton uses a sense-amplifier cycling technique to read out multiple bits in one time slot, thus significantly reducing input symbol match time [14]. Based on the described NOR, AND, and XOR operations, Compute Cache extends the logical operations by slightly modifying the SA design in [11]. In this paper, we utilize the XOR functionality presented in [11], and slightly modify it (similar to [15]) to be able to perform the shift operation (required for the proposed fused *ShiftRows* and *MixColumns* stages) in place without the need to store the intermediate result back into the subarray, thus, reducing processing cycles.

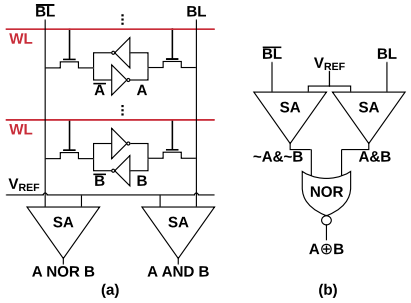

Fig. 2: The XOR bitline operation using 6T SRAM cells.

C. Threat Model

Similar to the state-of-the-art secure memory architectures [3], we assume the only trusted component is the processor chip and we do not expand the TCB of the system. Therefore, bus and memory are vulnerable and untrusted. We assume the attacker has physical access to the system and can snoop the bus, or scan the memory. This work proposes high-performance encryption for protecting data confidentiality. Addressing timing channels, replay attacks, and access pattern leakage threat models are left for future work.

III. Related Work

Hämäläinen et al. [16] (EE-1) and Mathew et al. [5] (EE-2) propose on-die AES encryption engines. EE-2 computes the entire AES round in composite-field arithmetic, resulting

in delay reduction in the critical S-box unit. They also use folded datapath design to reduce the wiring complexity of *ShiftRows* permutations. Wang et al. propose DW-AES [17] by utilizing the domain wall nanowire to implement the AES encryption algorithm in NVM. Though the throughput of DW-AES is 3.6× higher than the CMOS ASIC proposed in [18], the latency for one data block is still not low enough to accommodate real-time encryption.

To improve the performance of memory encryption, recent works propose in-memory encryption architectures. Xie et al. [6] present AIM, an efficient in-NVM implementation of AES. AIM provides bulk encryption/decryption for mobile devices to protect memory content only when the device is shut down, locked, or in sleep mode. Therefore, AIM does not provide real-time memory encryption to protect against bus and memory attacks. Aga et al. propose InvisiMem [7], which expands the trust base to the logic layer of 3D stacked memory to implement encryption. InvisiMem guarantees confidentiality, integrity, and protecting access patterns using a packetized interface and authentication to establish a secure communication channel between processor and memory. *Sealer* does not expand the TCB of the system and is the first work that proposes in-SRAM encryption for data confidentiality.

Commercial examples of memory protection solutions are Memory Encryption Engine (MEE) [3] in Intel SGX [1] and AMD's Secure Memory Engine (SME) [4]. The hardware component of SGX (i.e., MEE) protects the confidentiality, integrity, and freshness of processor-DRAM traffic for secure enclaves. The confidentiality protection in MME and SME are implemented by AES-128 encryption. *Sealer* provides the MEE- and SME-equivalent guarantee for memory confidentiality, but potentially with higher performance and lower area overhead by exploiting intrinsic parallelism and bitline computational capability of SRAM subarrays.

We compare the performance of *Sealer* with EE-1 [16], EE-2 [5], DW-AES [17], and AIM [6] in Section VI. We are not able to compare the performance of *Sealer* with Intel's MEE, AMD's SME, and Invisimem mainly because these designs provide other memory protections, such as data integrity. Therefore, we are not able to isolate the encryption performance from other components in their designs.

IV. Implementation

A. Data Organization

In this paper, an efficient implementation of the AES algorithm is accomplished using bitline computing of SRAM arrays, and can be realized by re-purposing a portion of the L3 cache or by replacing the existing on-chip encryption hardware. To efficiently utilize bitline computing, we organize the S-box, plaintext, keys, and intermediate data required by the AES algorithm in the same subarray, as shown in Figure 3(a). We assume 256×256 SRAM subarrays, following Intel's SandyBridge L3 cache structure [19]. As depicted in Figure 3, each subarray consists of 6 *Tiles*, where each *Tile* can store 51 data blocks ($Row[0] - Row[203]$ - each data block requires four rows), one set of round keys ($Row[204] - Row[247]$), and intermediate data required for *MixColumns* stage ($Row[248] - Row[253]$). The first 8 columns of the *Tile* are used to hold the S-box. All the data blocks arranged in the same set of rows can encrypt/decrypt data in parallel (i.e., 6 data blocks per subarray). The data blocks in the same *Tile* share the same bitlines, thus, cannot be computed in parallel.

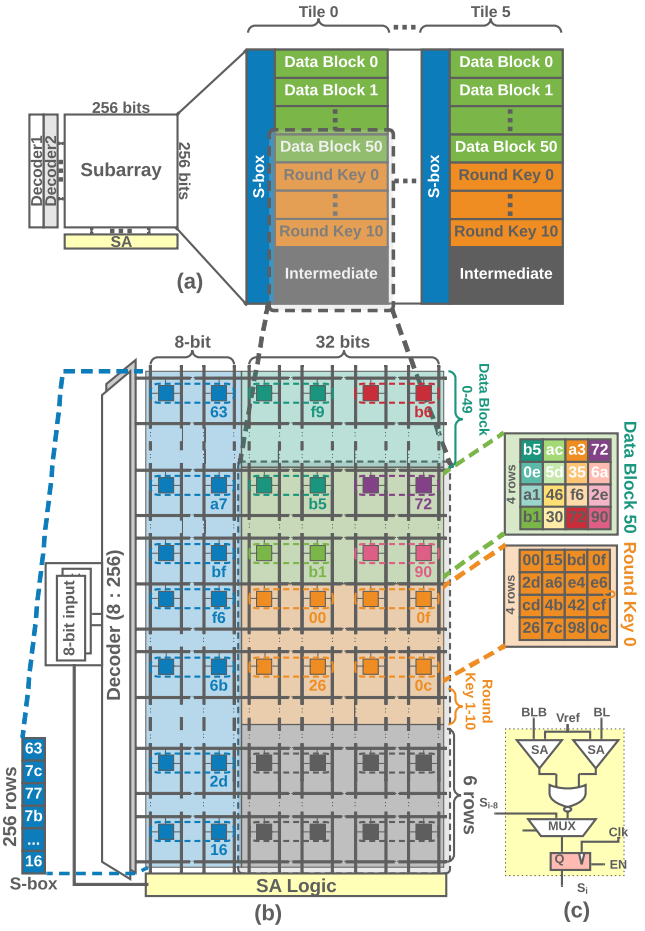

Fig. 3: (a) Data organization in one subarray. (b) The detailed structure of the first S-box and data (Tile 0) with three inputs of AES encryption algorithm: S-box (256-row, 8-bit), 128-bit plaintext and 128-bit cipher key. (c) The structure of the sense amplifier to support XOR and shift operations.

The S-box is originally a 16×16 matrix where each element is an 8-bit data. To avoid adding the extra LUT logic in the SRAM structure for S-box substitution in the *SubBytes* stage, and also to eliminate the communication and wiring overhead between the data block subarrays and the LUTs, *Sealer* proposes to re-organize the S-box into a matrix of 256 rows where each row has an 8-bit element (as shown in Figure 3(b) - the blue matrix) and utilize the same subarray as the plaintext and round keys are located. The memory decoder is used to decode the 8-bit input data to select the corresponding element for the substitution. This brings two advantages in *Sealer* compared to the prior work, AIM [6],

from the architectural aspect; (1) all the required data for performing the encryption is within the same subarray, and this brings the opportunity to fuse computation in different stages while the data is read into SA, and also perform intermediate data generation on-the-fly without the need to write them back into the subarray, thus, reducing the total number of cycles, and (2) unlike AIM that requires additional modules, such as LUTs and multiplexers, we repurpose the existing resources (i.e., memory array and peripherals) to perform the required computation, and thus, avoiding extra hardware overhead. Details are discussed in Sections IV-C and IV-D.

B. AddRoundKey

AddRoundKey receives two matrices, the data matrix (i.e., the plaintext) and the key matrix (i.e., the round key). Each corresponding byte in the data matrix is bit-wise XORed with each byte in the key matrix by activating two rows using two decoders. We follow the bitline XOR functionality implementation in [6], which costs 3 times longer than a single subarray read/write access.

As depicted in Figure 3 (b), the data matrix (green area) and the key matrices (orange area) rows are arranged in consecutive rows (column-aligned) in the subarray, and the elements within the same row in the matrix are located next to each other horizontally in the subarray (i.e., each 128-bit data matrix or 128-bit round key requires 4 consecutive rows and 32 consecutive columns). This means that *Sealer* can perform the bitwise XOR in parallel on 192 bits (4 elements in one row of a *Tile*, across 6 *Tiles* in a subarray) of plaintext within one subarray in only 3 cycles (note that several subarrays can work all in parallel without almost any performance overhead!). The *AddRoundKey* output (i.e., the bitwise XOR) is sensed in the SAs (Figure 3 (c)) by activating the equivalent rows in the data matrix and key matrix. To maximally utilize the data that is already in the sense amplifier (i.e., row buffer hit) and eliminate unnecessary write cycles, we perform the computation of the next stage (i.e., *SubBytes*) on the output of the first row before writing it back to the subarray.

C. Fused SubBytes and ShiftRows

To efficiently implement the *SubBytes* and *ShiftRows* stages and reduce the total number of processing cycles, we fuse the *ShiftRows* and *SubBytes* stages. This is done by selecting the right order of the elements from the output of *AddRoundKey* and feeding the element as the input address of the decoder to read the substitution value. The order is determined by the number of shifts required in the *ShiftRows* stage.

AES defines a substitution box (S-box), which consists of a 16×16 byte matrix and is used to substitute each byte of the data block with the corresponding byte in the S-box matrix by using the four MSB bits to select one of the 16 rows and the four LSB bits to choose one of the 16 columns. Instead of adding LUTs and incurring the extra hardware overhead, *Sealer* re-purposes the existing resources in SRAM and store the S-box in the same subarray as data block and keys are stored. We accordingly reorganize the conventional 16×16 byte matrix to a matrix of 256×1 where each row represents one byte (Figure 3 - blue S-box matrix). We then use the result of *AddRoundKey* stage, which is already stored in the sense amplifiers, as the input of the decoder and read the substitution bytes consecutively. The input of the decoder is equipped with a two 8-bit entry FIFO buffer, to enable the in-place substitution (*SubBytes*) and shifting (*ShiftRows*).

Figure 4 demonstrates how the proposed *fused SubBytes and ShiftRows* stage works for the third row in the data bock matrix using a step-by-step example. In the initial time step (T0), the XOR results of the third row in the data matrix (a1, 46, f6, 2e) and the third row of the round key matrix (cd, 4b, 42, cf) are written into the SA buffers (6c, 0d, b4, e1). We first explain the conventional computation used in prior work (i.e., computing *SubBytes* and *ShiftRows* consecutively), and then, we describe the proposed fused approach in *Sealer*.

Assume that the substitution bytes (i.e., the output of the *SubBytes* stage) for (6C, 0d, b4, e1) are (50, d7, 8d, f8). The output of the *SubBytes* will be the input of *ShiftRows* in the next stage. Because the computation is performed on the third row, the *ShiftRows* stage will shift the byte to the left two times using cyclic shift, i.e., the output of the *ShiftRows* stage will be (8d, f8, 50, d7).

In the proposed fused approach, we first select the element that its corresponding substitution byte will be located on the right-most side of the array after the *ShiftRows* stage is completed, i.e., $sel = 10(\text{“}0d\text{”})$ will be read into the input buffer. Then, we select the elements in the *AddRoundKey* output one by one according to their corresponding locations in the *ShiftRows* output from the right-most side element to the left-most side element in the array, i.e., $sel = 10(0d) \rightarrow 11(6c) \rightarrow 00(e1) \rightarrow 01(b4)$.

For the correct functionality, the input buffer should have two entries. In T1, 0d and 6c are read to the FIFO buffer. Then, the decoder takes 0d as the input address and the corresponding substitution byte is read into the row buffer (i.e., d7). In T2, first, e1 is selected (Sel = 00) and read into the second entry of the input buffer. Then, the intermediate output is shifted to the right by one byte, and finally, the decoder takes 6c as the input and the corresponding substitution byte is read into the row buffer (i.e., 50). In T3, first, b4 is selected (Sel = 01) and read into the second entry of the input buffer. Then, the intermediate output is shifted to the right by one byte, and finally, the decoder takes e1 as the input and the corresponding substitution byte is read into the row buffer (i.e., f8). In T4, first, the intermediate output is shifted to the right by one byte, and then, the decoder takes b4 as the input and the corresponding substitution byte is read into the row buffer (i.e., 8d). Finally, in T5, the intermediate buffer is shifted to the right by one byte and the final output of the *ShiftRows* is generated and ready to be used by the *MixColumns* stage.

The shift operation is implemented by introducing a latch and a multiplexer within the SA [15], as shown in Figure 3(c). The proposed fusion approach greatly reduces write cycles, thus reducing processing latency (evaluated in Section VI-B). It is important to note that the fused *SubBytes* and *ShiftRows*

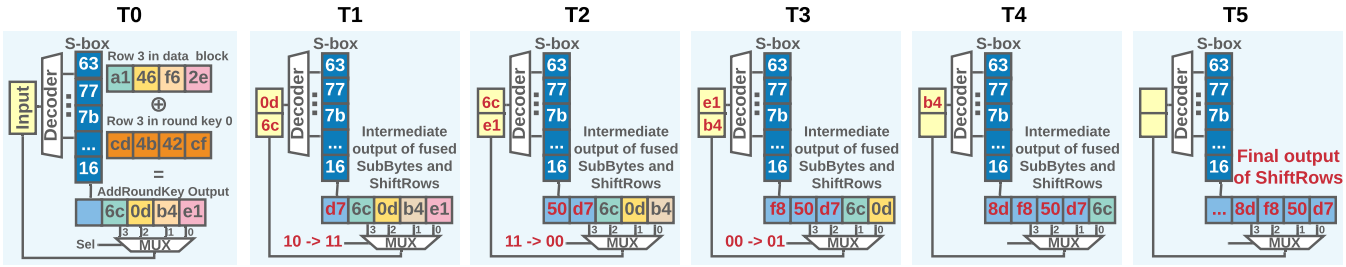

Fig. 4: Fusion of the *SubBytes* and *ShiftRows* stages for the third row of the data matrix.

computation are done in parallel across all the *Tiles* (i.e., one data block in each *Tile* as the data blocks within one *Tile* cannot be processed in parallel) in and across subarrays.

D. MixColumns

In the *MixColumns* stage, each column of the input data matrix (i.e., the output matrix of *ShiftRows* stage) is multiplied with a two-dimensional constant array, called the fixed matrix, to obtain the corresponding output column. All the addition and multiplication are both defined over a finite field. To achieve efficient computation in SRAM, and to reduce the resource consumption and the number of SRAM accesses for intermediate results, we decompose the matrix multiplication of data columns and the fixed matrix into matrix elements, an intermediate value T_c, and the product of matrix elements, following [6]. The decomposition equation is as follows:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} B_{0,c} \\ B_{1,c} \\ B_{2,c} \\ B_{3,c} \end{bmatrix} = \begin{bmatrix} 2B_{0,c} + 3B_{1,c} + B_{2,c} + B_{3,c} \\ B_{0,c} + 2B_{1,c} + 3B_{2,c} + B_{3,c} \\ B_{0,c} + B_{1,c} + 2B_{2,c} + 3B_{3,c} \\ 3B_{0,c} + B_{1,c} + B_{2,c} + 2B_{3,c} \end{bmatrix} = \begin{bmatrix} T_c \oplus 2B_{0,c} \oplus 2B_{1,c} \oplus B_{0,c} \\ T_c \oplus 2B_{1,c} \oplus 2B_{2,c} \oplus B_{1,c} \\ T_c \oplus 2B_{2,c} \oplus 2B_{3,c} \oplus B_{2,c} \\ T_c \oplus 2B_{0,c} \oplus 2B_{3,c} \oplus B_{3,c} \end{bmatrix} \quad (1)$$

where $B_{r,c}$ denotes the byte in row r and column c, and T_c is the intermediate result for column c. T_c is calculated as:

$$T_c = B_{0,c} \oplus B_{1,c} \oplus B_{2,c} \oplus B_{3,c}. \quad (2)$$

The computation of $2 \times B_{r,c}$ elements in Equation 1 is done in *ShiftRows* stage by shifting the output of *ShiftRows* stage to the left by one bit and storing them back in the intermediate region of the subarray. Figure 5(a) shows how T_c in each column is calculated and stored in the subarray in 6 steps using an example (note that the calculation of T_c is fully parallel in each column). To calculate T_0, in Step 1, the two (red) worldlines are activated simultaneously to calculate the XOR of $B_{0,0}$ and $B_{1,0}$, and store the result (I_0) in the intermediate region in Step 2. Next, the XOR of $B_{2,0}$ and $B_{3,0}$ is calculated in Step 3 and written back to the array (I_1) in Step 4. Finally, T_0 is calculated (Step 5) and written back (Step 6). Figure 5(b) shows how the final output is calculated for $T_c \oplus 2B_{0,c} \oplus 2B_{1,c} \oplus B_{0,c}$ by activating the corresponding rows, calculating the XOR of the activated cells, and writing the intermediate results back. After 6 steps, the output of the *MixColumn* ($B'_{0,0}$) overwrites the original data ($B_{0,0}$).

V. Key Generation and Storage

Similar to state-of-the-art memory encryption engines [3], we assume the encryption key is generated using a hardware

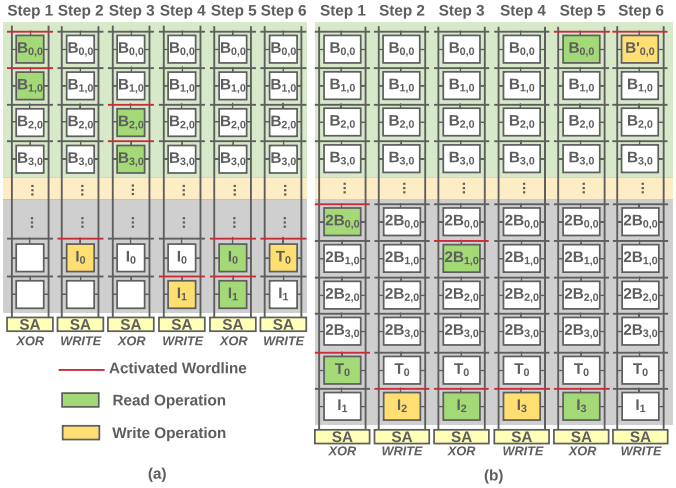

Fig. 5: Example of *MixColumns* stage.

random number generator and implemented in the processor chip. The key is inaccessible outside of the chip. The expansion of the given cipher key can also be utilized to obtain 11 partial keys, which are used in the initial round, the 9 main rounds, and the final round. The expansion process can be implemented in the subarray, including shift operations between columns, the *SubBytes* stage, the XOR operations between rows and a given reconstruction matrix. The generated keys will be stored within the subarray as shown in Figure 3.

VI. Evaluation

A. Evaluation Methodology

In this section, we evaluate the performance, throughput-per-area and energy/power of *Sealer* and compare it to prior in-memory encryption solutions [6], [17] and on-chip memory encryption implementations [5], [16]. AIM architecture [6] is implemented in-MRAM (*AIM-NVM*) and *Sealer* utilizes SRAM-based subarrays. To decouple architectural contribution from the technology contribution and have an apples-to-apples comparison between AIM and *Sealer*, we model and evaluate the architecture of AIM in SRAM (*AIM-SRAM*). The area overhead evaluation presented in the AIM is based on relative numbers. To directly compare the area overhead of AIM-NVM and AIM-SRAM with *Sealer*, we use NVSim [20] to obtain the absolute area parameters for the peripherals used in AIM-NVM. The read/write access latency to a 256×256 6T SRAM

subarray is 163 ps, and an XOR operation in SRAM costs 489 ps, which are extracted from the SPICE simulations with 28nm SOI CMOS process [11], [14]. To evaluate the energy and power consumption of SRAM-based and NVM-based designs, we use DESTINY simulator [21].

The baselines for comparison against *Sealer* are (1) EE-1 [16], which introduces an AES encryption based on a dedicated engine for low power consumption at 290 MHz, (2) EE-2 [5], which presents a high-frequency integrated circuit encryption engine at up to 2.13 GHz, (3) DW-AES [17], which uses domain-wall nanowires to implement AES encryption in NVM at 30MHz, (4) AIM-NVM, and (5) AIM-SRAM.

B. Latency Analysis

Figure 6 compares the data encryption latency of different solutions normalized to the *Sealer* latency for 24 data blocks (384 bytes) and for 192 data blocks (3072 bytes, which is approximately the size of one cache slice in L3 [19]). *Sealer* has 30× (243×), 1.22× (9.8×), and 1880× (15040×) lower latency than EE-1, EE-2, and DW-AES, respectively, for encrypting 24 data blocks (192 data blocks). For all these solutions, *Sealer's* performance improves significantly when increasing the memory capacity because of the intrinsic bit-level and subarray-level parallelism in memory. For example, EE-2 can operate on only four blocks in parallel, while *Sealer* can encrypt 192 data blocks simultaneously when utilizing the 2MB SRAM. In general, in-memory solutions (*Sealer*, AIM-NVM, and AIM-SRAM) provide significantly higher parallelism than dedicated hardware engines (EE-1, and EE-2).

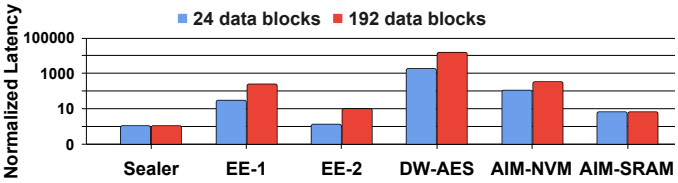

Fig. 6: Latency comparison among different baselines normalized to *Sealer* for encrypting 24 and 192 data blocks.

Sealer has almost 6.5× lower latency than AIM-SRAM (architectural contribution of *Sealer* compared to AIM). This is because the required number of cycles for encrypting six data blocks in a subarray in AIM-SRAM is almost 6.5× higher than *Sealer*, as shown in Figure 7. Figure 7 compares the cycle breakdown for different stages in *Sealer* and AIM. *AddRoundKey* stage has the same number of cycles in both architectures. However, the fused *SubBytes* and *ShiftRows* stage in *Sealer* requires 59.5% less cycle than the total cycles for *SubBytes* and *ShiftRows* stages in AIM. This benefit is enabled by storing all required data (i.e., data blocks, S-box, round keys, and intermediate data) in the same sub-array, which facilitates computation fusion among different stages, thus, resulting in fewer write cycles. Moreover, in the *MixColumns* stage, unlike AIM which uses additional LUTs, *Sealer* utilizes the existing rows and peripherals in the under-processed subarray to store intermediate results, thus, avoiding the bandwidth bottleneck of moving data between subarrays and limited LUTs. Increasing the number of LUTs can provide more parallelism in AIM, however, it significantly increases area overhead, thus, decreasing throughput-per-area.

Sealer has 107× and 323× lower latency than AIM-NVM for encrypting 24 and 192 data blocks, respectively. The latency reduction for encrypting 24 data blocks comes from the architectural contribution (about 6× due to the data layout and fusing computational stages) and technology contribution (about 18× due to the choice of SRAM compared to NVM) in *Sealer*. Overall, *Sealer* presents a low latency encryption solution, which makes it suitable for real-time encryption.

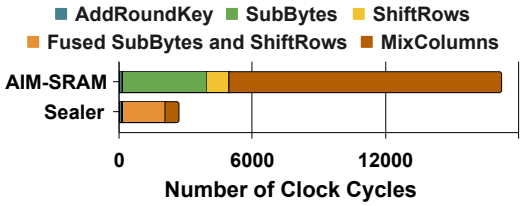

Fig. 7: Breakdown of processing cycles in AIM-SRAM and *Sealer* for encrypting 6 data blocks.

C. Throughput Normalized to Area

Figure 8 compares the throughput-per-area for *Sealer*, AIM-NVM, and AIM-SRAM. *Sealer* has 7.2× higher throughput per unit area than AIM-SRAM. These benefits come from the fact that (1) *Sealer* only incurs negligible extra overhead to the SRAM arrays by sharing the memory resources and peripheral among different stages of computation and communication, while AIM-SRAM requires additional hardware components, such as LUTs and bundles of MUX/DEMUXes, and (2) *Sealer* utilizes an efficient algorithm/architecture methodology to fuse different computational stages, which maximizes on-the-fly computation and minimizes data movement. *Sealer* can also achieve more than 91× throughput per unit area compared to the AIM-NVM. Although the area consumption of AIM-NVM is lower than *Sealer*, the frequency of *Sealer* is 133× higher than AIM-NVM; thus, results in higher throughput.

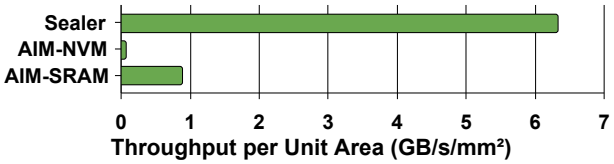

Fig. 8: Throughput comparison among AIM-NVM, AIM-SRAM, and *Sealer* for encrypting 192 data blocks.

D. Energy/Power Analysis

Figure 9 compares the energy and power of Sealer, AIM-NVM, and AIM-SRAM when encrypting 24 and 192 data blocks. To complete the encryption, Sealer consumes 3× less energy compared to AIM-NVM and AIM-SRAM, thanks to our fusion scheme and the reduction of LUT queries. The

energy consumption of AIM-SRAM is slightly lower than AIM-NVM. This is because the *MixColumns* stage, which has the highest number of operations during encryption in the AIM architecture, contains a large number of write operations (the write dynamic energy of AIM-NVM is higher than the read dynamic energy). Sealer has 34× and 2× higher power consumption compared to AIM-NVM and AIM-SRAM, respectively. This is because *Sealer* has a higher operational frequency and also maximizes computational parallelism so that almost all units are activated and computing, which greatly increases compute/memory unit utilization. However, the other two designs have all other computing and memory units idle when querying the lookup table.

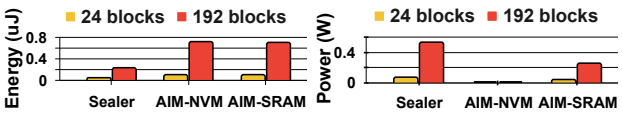

Fig. 9: Energy & power comparison among *Sealer*, AIM-NVM, and AIM-SRAM for encrypting 24 and 192 data blocks.

VII. Conclusion

In this paper, we propose *Sealer*, a real-time low-overhead in-SRAM AES engine to encrypt the plaintext on a CPU chip with narrowed trusted computing base. By efficiently mapping the algorithm to the *Sealer* architecture at a finer granularity, data movement is significantly reduced. Our evaluation results show significant performance (up to 323×) and throughput-per-area (up to 91×) improvement over the state-of-the-art in-memory and specialized encryption engines. Future work will extend *Sealer* to provide data integrity and access pattern protection in addition to data confidentiality.

References

[1] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, 2016.

[2] J. A. Halderman *et al.*, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, 2009.

[3] S. Gueron, "A memory encryption engine suitable for general purpose processors," Cryptology ePrint Archive, 2016.

[4] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," 2016.

[5] S. Mathew *et al.*, "53Gbps native GF$(2^4)^2$ composite-field AES-encrypt/decrypt accelerator for content-protection in 45nm high-performance microprocessors," in *VLSIC*, 2010.

[6] M. Xie *et al.*, "Securing emerging nonvolatile main memory with fast and energy-efficient AES in-memory implementation," *TVLSI*, 2018.

[7] S. Aga and S. Narayanasamy, "InvisiMem: Smart memory defenses for memory bus side channel," in *ISCA*, 2017.

[8] S. Angizi, Z. He, and D. Fan, "PIMA-Logic: A novel processing-in-memory architecture for highly flexible and energy-efficient logic computation," in *DAC*, 2018.

[9] S. Angizi *et al.*, "RIMPA: A new reconfigurable dual-mode in-memory processing architecture with spin hall effect-driven domain wall motion device," in *ISVLSI*.

[10] S. Jeloka *et al.*, "A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory," *IEEE JSSC*, 2016.

[11] S. Aga *et al.*, "Compute caches," in *HPCA*, 2017.

[12] C. Eckert *et al.*, "Neural cache: Bit-serial in-Cache acceleration of deep neural networks," in *ISCA*, 2018.

[13] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *ISCA*, 2019.

[14] A. Subramaniyan *et al.*, "Cache automaton," in *MICRO*, 2017.

[15] A. Nag *et al.*, "Gencache: Leveraging in-cache operators for efficient sequence alignment," in *MICRO*, 2019.

[16] P. Hamalainen *et al.*, "Design and implementation of low-area and low-power AES encryption hardware core," in *DSD*, 2006.

[17] Y. Wang *et al.*, "DW-AES: A domain-wall nanowire-based AES for high throughput and energy-efficient data encryption in non-volatile memory," *IEEE TIFS*, 2016.

[18] S. Mathew *et al.*, "340 mV–1.1 v, 289 Gbps/W, 2090-Gate NanoAES hardware accelerator with area-optimized Encrypt/Decrypt GF$(2^4)^2$ polynomials in 22 nm tri-gate CMOS," *IEEE JSSC*, 2015.

[19] O. Lempel, "2nd generation Intel® core processor family: Intel® core i7, i5 and i3," in *HotChips*, 2011.

[20] X. Dong *et al.*, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE TCAD*, 2012.

[21] M. Poremba *et al.*, "Destiny: A tool for modeling emerging 3d nvm and edram caches," in *DATE*, 2015.