AIM: Fast and Energy-Efficient AES In-Memory Implementation for Emerging Non-volatile Main Memory

Mimi Xie*, Shuangchen Li†, Alvin Oliver Glova†, Jingtong Hu*, Yuangang Wang‡, Yuan Xie†
*Department of Electrical and Computer Engineering, University of Pittsburgh, Pittsburgh, PA 15261, USA
†Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, USA
*Huawei Technologies Co., Ltd. P.R. China
Email: *{mix17, jthu}@pitt.edu, †{shuangchenli, aomglova, yuanxie}@ece.ucsb.edu, ‡wangyuangang@huawei.com

***Abstract*—Non-volatile main memory-based systems pose an opportunity for an attacker to readily access sensitive information on the memory because of its long retention time. While real-time memory encryption with dedicated AES engine can address this vulnerability, it incurs extra performance and energy overheads. As an alternative, we propose an AES in-memory implementation, AIM, to encrypt the whole/part of the memory only when it is necessary. We leverage the benefits offered by the in-memory computing architecture to address the challenges of the bandwidth intensive encryption application. We take advantage of NVM's intrinsic logic operation capability to implement the AES task. Embracing the massive parallelism inside the memory, AIM outperforms existing mechanisms with higher throughput yet lower energy consumption. Compared with state-of-the-art AES engine running at 2.1GHz, AIM can speed up the encryption process by 80× for a 1GB NVM.**

I. Introduction

DRAM has been employed as the main memory for decades. However, as technology scales down, future DRAM will suffer from prohibitively high leakage power. Consequently, researchers are actively developing promising candidates such as phase change memory (PCM) [16], resistive random access memory (ReRAM) [14], and spin-transfer torque magnetic random access memory (STT-MRAM) [11] as next-generation non-volatile main memory (NVMM). NVMM has several significant advantages over traditional DRAM main memory such as non-volatility, high density, and low leakage power. The nature of non-volatility avoids the need of frequent refresh for DRAM and allows the data to be retained a long time after power is off. Intel's recent announcement of 3D Xpoint [2] and JEDEC's NVDIMM-P specification [1] are some latest efforts towards the goal of next-generation NVMM.

In spite of these advantages, NVMM suffers from a new security vulnerability. Since the information in NVMM will not lose data after power is turned off, an attacker with physical access to the system can readily scan the main memory content and extract all valuable information from the main memory [4]. In contrast, the security of DRAM memory relies on its short retention time which varies from 500 ms to 50 seconds [12]. To protect the data of the NVMM, the whole memory should be provided with a security mechanism with comparable security level to DRAM.

Real-time memory encryption is an effective solution for this vulnerability, in which every cache line is encrypted/decrypted before being written/read from the main memory. Unfortunately, decryption latency on the critical path causes significant performance loss. In addition, encryption/decryption at every memory access results in severe energy overhead. i-NVMM [4] encrypts main memory incrementally and maintains an unencrypted working set for quick access. However, such a strong real-time protection is not always necessary. For example, the attack to mobile devices (e.g. smart phone or laptop) requires physical access to the NVMM which rarely happens. Memory encryption is required only when the device is shut down or put into sleep/screenlock mode. Thus, instead of real-time encryption, another approach that encrypts the whole/most part of the memory only when it is necessary is preferred in such mobile scenarios.

However, one-time memory encryption approach still faces two challenges: First, it should be fast enough to maintain a low vulnerability window when locked and provide instant response when unlocked. Second, it should be energy-efficient considering the limited battery life. This paper proposes AIM, a novel AES in-memory encryption architecture for fast and energy-efficient NVMM encryption. Instead of integrating a dedicated AES engine on the memory side, we solve the security problem originally raised by non-volatility, with the non-volatility itself. To this end, we leverage the non-destructive read in NVMs for performing efficient XOR operations, which dominate AES. After adding lightweight logic gates to the memory peripherals circuitry, we can perform the entire AES procedure in-place. Embracing the benefit of processing-in-memory (PIM) architecture, AIM takes advantage of large internal memory bandwidth, vast bitline-level parallelism, and low in-situ computing latency.

To the best of our knowledge, this is the first work that exploits the intrinsic logic operation capability of NVMs to implement in-memory encryption for NVMs. We explore different levels (chip, bank, and subarray) of parallelism to provide different design choices for different performance and energy efficiency requirement.

This work was partially supported by NSF CCF-1500848, CCF-1719160, CCF-1527506, and CNS-1464429.

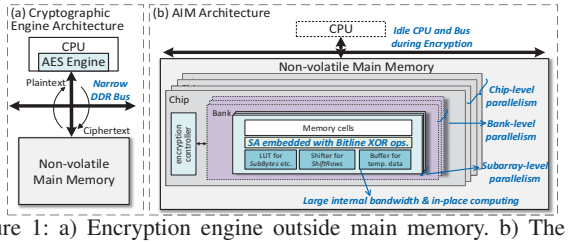

Figure 1: a) Encryption engine outside main memory. b) The proposed in-memory encryption design.

II. Related Work and Overview

There have been several work on encryption of NVM [4], [15]. These implementations perform encryption/decryption incrementally which rely on a dedicated encryption engine on the processor or in the main memory. Different from them, [5] proposes to perform one-time encryption for mobile devices only when the device is screen-locked. In one-time memory encryption, the memory bandwidth is a bottleneck since a dedicated AES encryption engine provides a much larger throughput than the DDR throughput. [13], [9] propose PIM encryption based on domain wall memory (DWM).

The proposed PIM architecture, AIM, takes advantage of fast in-memory operations of NVMM in Pinatubo [8], which modifies the sense amplifier (SA) circuit so that the SA can carry out bitwise operations. To perform XOR in Pinatubo, two operand rows are opened sequentially. The two operands will go through the XOR circuit inside the modified SA. The readout of this SA is the XOR result of these two rows.

The mechanism of AIM is shown in Figure 1(b). Different from the co-processor AES encryption engine (EE) (Figure 1(a)), AIM avoids the narrow DDR bus and embraces the large intra-memory bandwidth. It also benefits from parallel encryption by leveraging three levels of parallelism inside the memory. An encryption controller is implemented in each chip to provide control signals to direct the encryption process.

III. AES In-Memory Implementation

A. Data Organization

In this work, in-memory encryption is performed directly on the data in the memory cells. These data are read out with SAs, each of which is shared by several adjacent columns with a MUX as shown in Figure 2. We distribute the 8 bits of each element in the data matrix into different mats and different columns in the same mat so that they can be used concurrently. In this way, the plaintext data block does not have to be pre-transformed into matrix form before encryption starts.

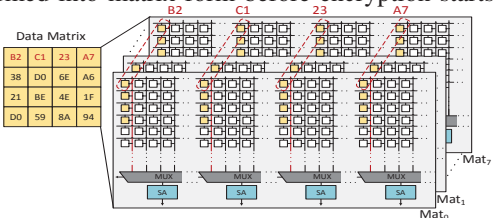

Figure 2: Distributed data organization for AES encryption.

Figure 2 illustrates the memory distribution for one data matrix, which is distributed into eight mats so that each mat has 1-bit level of the data matrix. To encrypt each row of the data matrix in parallel, four columns of the data matrix are distributed to different columns of memory array connecting four adjacent SAs separately. These four columns of the memory array are of the same local column address. In this way, when one row of the subarray is activated and a local column address is selected for each MUX, every four adjacent SAs will sense out a row of the data matrix.

B. AddRoundKey

In this stage, each byte of the data matrix is combined with the corresponding subkey of the key matrix. *AddRoundKey* is implemented with the modified SA design of Pinatubo [8].

Figure 3 shows the *AddRoundKey* transformation for one row of data. First, the first row of the data matrix is read into the capacitor in each SA by activating the first wordline in red color and selecting a column with MUX. Second, the first row of the key matrix is read into the latch in each SA by activating the second wordline in red color and selecting a column with MUX. Then, the XOR result of two rows is latched in each SA. This *AddRoundKey* transformation is parallelized because of multiple SAs. In our design, after *AddRoundKey* transformation for one row of data, SubBytes is immediately performed for this row of data instead of continuing performing *AddRoundKey* for all four rows.

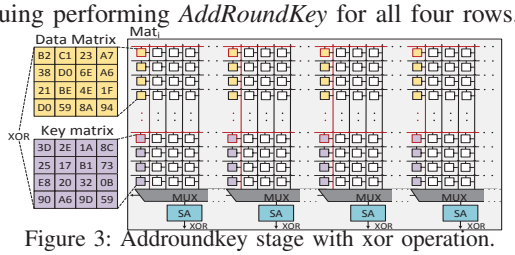

Figure 3: Addroundkey stage with xor operation.

The initial *AddRoundKey* stage is performed with the initial key. The other 10 rounds are performed with the corresponding round key. The key generator shared among all memory chips generates the key and sends it to different chips. After each round of encryption, this key generator expands the initial key to get the round key. As shown in Figure 3, the encryption key is maintained in the non-volatile memory array and round keys overwrite the encryption key after each round of encryption.

C. SubBytes

In this step, each byte of the data matrix is replaced with a new byte with S-box to obscure the relationship between the key and the ciphertext. In this paper, S-box is realized with look-up tables (LUT) by implementing combinational logic which has 8-bit input and 8-bit output since it incurs lower overhead compared with ROM.

After the *AddRoundKey* stage of one row of state matrix, the intermediate results are latched in the SAs. For *SubBytes* transformation, each byte of the data matrix is decoded from eight mats and input to the S-box as shown in Figure 4. In this figure, the *AddRoundkey* results of the second row of data matrix are latched in the SAs. *SubBytes* is performed on the second byte $C7$. The output of S-box is the substituted byte $C6$. In this figure, there is one S-box combinational logic which has 8-bit input and 8-bit output. Since we can only input one byte each time to the S-box, the *SubBytes* transformation can only be done sequentially which takes a long time. To accelerate the *SubBytes* transformation, we can add more S-box combinational logic to enable parallel *SubBytes* operation.

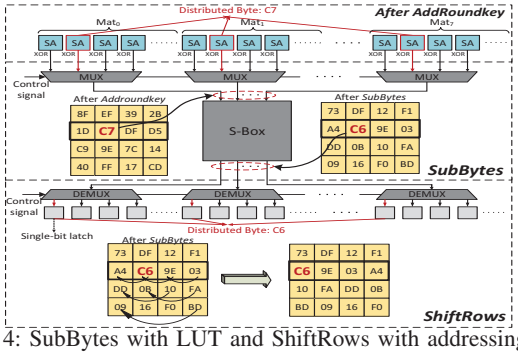

Figure 4: SubBytes with LUT and ShiftRows with addressing logic.

At the same time, we need to consider the hardware overhead introduced by multiple S-box. After we obtain the 8-bit output of S-box, it will not be immediately written back. Instead, the next stage, *ShiftRow*, will be performed on the output.

D. ShiftRows

In this step, while the top row remains unchanged, each bit in the left three rows of the bit-level data matrix is cyclically shifted left by 1, 2 and 3 bits (right by 1 bit), respectively.

ShiftRows transformation is realized with control signal and address decoding, as shown in the lower part of Figure 4. Originally, the 8-bit output of S-box needs to be written back where each input bit is located. This process needs address decoding to write to the right position. *ShiftRows* transformation can leverage this address decoding process to do shifting. By combining an offset with the column address, the output of S-box is shifted to another address according to the *ShiftRows* algorithm. In Figure 4, the second byte $C6$ in the second row of state matrix after *SubBytes* needs to be shifted to the left by one byte. This means each bit needs to be shifted left by one bit according to the data matrix distribution in the memory. This shifting process is done by selecting the first column with the control signal.

After *ShiftRows* transformation, each bit will be buffered in the single-bit latch until *SubBytes* and *ShiftRows* are performed on all data in the SAs. Then, the values in the row buffer are transmitted to the write driver and written back to the memory array. This row buffer gathers the intermediate results and avoids extra writes to the non-volatile memory row.

E. MixColumns

In *MixColumns* stage, the data matrix is multiplied by a known matrix. In this way, the four bytes of each column of the data matrix are combined together using an invertible linear transformation to provide diffusion in the cipher.

This matrix multiplication is done in the finite field GF(2^8), which can be decomposed to modular multiplication and XOR operations. We use $S_{i,j}$ and $S'_{i,j}$ to indicate the byte in row i, column j of the state matrix and the transformed state matrix respectively. The *MixColumns* transformation is as follows:

$$\begin{aligned} S'_{0,j} &= 2 \cdot S_{0,j} \oplus 3 \cdot S_{1,j} \oplus S_{2,j} \oplus S_{3,j}; \\ S'_{1,j} &= S_{0,j} \oplus 2 \cdot S_{1,j} \oplus 3 \cdot S_{2,j} \oplus S_{3,j}; \\ S'_{2,j} &= S_{0,j} \oplus S_{1,j} \oplus 2 \cdot S_{2,j} \oplus 3 \cdot S_{3,j}; \\ S'_{3,j} &= 3 \cdot S_{0,j} \oplus S_{1,j} \oplus S_{2,j} \oplus 2 \cdot S_{3,j}; \end{aligned} \tag{1}$$

Multiplication-by-2 (M-2) in the finite field can be realized by leveraging an LUT. M-3 in the finite field GF(2^8) of *MixColumn* can be realized with M-2 and XOR logic. This is because

$$3 \cdot S_{i,j} = 2 \cdot S_{i,j} \oplus S_{i,j} \tag{2}$$

Therefore, *MixColumns* stage is decomposed into M-2 LUT and XOR operations. *MixColumns* generates several intermediate values. To both accelerate this transformation and maintain a low hardware overhead, we leverage the vacant non-volatile memory rows as buffer rows for intermediate results. The *MixColumns* stage is realized with LUT and XORs as follows:

$$\begin{aligned} S'_{0,j} &= T_j \oplus \underline{2 \cdot S_{0,j}} \oplus \underline{2 \cdot S_{1,j}} \oplus S_{0,j} \\ S'_{1,j} &= T_j \oplus \underline{2 \cdot S_{1,j}} \oplus \underline{2 \cdot S_{2,j}} \oplus S_{1,j} \\ S'_{2,j} &= T_j \oplus \underline{2 \cdot S_{2,j}} \oplus \underline{2 \cdot S_{3,j}} \oplus S_{2,j} \\ S'_{3,j} &= T_j \oplus \underline{2 \cdot S_{0,j}} \oplus \underline{2 \cdot S_{3,j}} \oplus S_{3,j} \end{aligned} \tag{3}$$

where

$$T_j = S_{0,j} \oplus S_{1,j} \oplus S_{2,j} \oplus S_{3,j}; \tag{4}$$

The first step of *MixColumns* is M-2 transformation. This process shares the same address decoding logic of S-box with a MUX as shown in Figure 5. Since only one byte can be input each time to the LUT, this transformation is done sequentially. To accelerate this process, multiple M-2 LUTs are added to enable parallel operations. Like the S-box design, different multiplication-by-2 LUT designs are considered with different encryption speed and overhead. After M-2 transformation, outputs are latched in a row buffer until all bytes of the activated row finishes M-2 transformation. Then data in this row buffer is written to a vacant memory row.

The next step of *MixColumns* is calculating T_j following Eq. (4). Every time two rows are activated to get the XOR result of two memory cells, this result is written to an empty buffer row. In this step, since all SAs are working simultaneously, T_j for each column is calculated in full parallel.

The final step of *MixColumns* is calculating the result of *MixColumns* transformation following Eq. (3). In this step, with M-2 LUT results stored in four rows and T_j values, *MixColumns* for one row of selected columns can be completed in six steps. Four operands are XORed together to get $S'_{0,j}$ and then this result is written back replacing $S_{0,j}$.

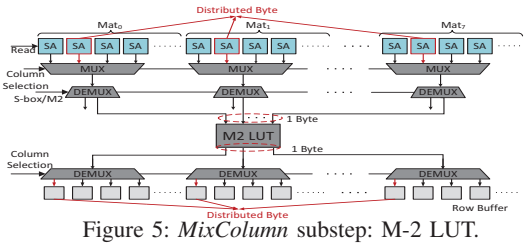

Figure 5: *MixColumn* substep: M-2 LUT.

IV. Experimental Evaluation

A. Experiment Setup

AIM is evaluated on both MRAM-based and PCM-based main memory with a DDR3 interface and 65nm technology. The page size of MRAM-based and PCM-based main memory is 512 bit and 1024 bit, respectively. We conservatively assume the capacity of MRAM is 256Mb per chip with a $34F^2$ cell size and the capacity of PCM is 1Gb per chip with the cell size of $9F^2$. We modified NVSim [6] and CACTI-3DD [3] to achieve the parameters for the NVM-based main memory.

Three configurations of AIM are compared with three dedicated memory encryption engines (EEs). The configurations

of AIM are: **AIM** has only one bank working on encryption at one time. **AIM-B** has the encryption add-on circuit for each bank and all banks in a chip can work in parallel. **AIM-S** has the add-on circuit for each subarray and multiple subarrays in the same bank work simultaneously. The dedicated EEs are: **EE-1** [7] is an AES encryption core with low frequency. **EE-2** [10] implements an AES CMOS ASIC encryption core with high frequency. **DW-AES** [13] implements an AES encryption core with domain-wall nanowires.

B. Performance and Energy Evaluation

1) Latency: Figure 6 shows the encryption latency of 1GB memory. **EE-2** has the fastest encryption speed among the three EEs. In **EE-2**, the encryption latency is offset by the writing latency because its write latency is larger than the encryption latency. For three AIM designs, higher levels of parallelism in NVMM achieves higher performance. When the subarray level parallelism is enabled, **AIM-S** is able to encrypt the whole memory in 0.33 seconds and 0.018 seconds for PCM and MRAM implementations, respectively. With PCM implementation, **AIM-B** has similar performance for 1GB main memory and **AIM-S** can encrypt 1GB much faster than **EE-2**. With MRAM implementation, all three designs of AIM work faster than **EE-2**. In addition, when the size of NVMM scales up, the latency of **EE-2** will scale up accordingly. However, for **AIM-B**, as long as main memory power budget allows, it can continue to leverage the parallelism and maintain a short encryption latency.

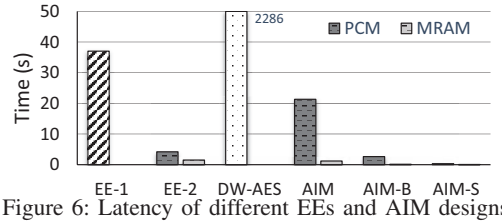

Figure 6: Latency of different EEs and AIM designs.

2) Energy Efficiency: Figure 7 (left) compares the energy efficiency for encrypting a 128-bit block. From this figure, **EE-2** incurs the smallest energy, while **AIM** ranks the third. Figure 7 (right) shows the energy consumption for encrypting 1GB NVMM. The lower parts of the first 6 columns show the energy spent on accessing main memory and the upper parts show the energy spent on encrypting process with the encryption engines. **EE-1**, **EE2**, and **DW-AES** cost significant amount of energy on memory access. This is because that the encryption processor needs to read a memory block from the main memory and then write this memory block back to its original position after encryption is completed. **AIM** costs the lowest energy compared with the three encryption engines because it encrypts memory blocks inside the NVMM.

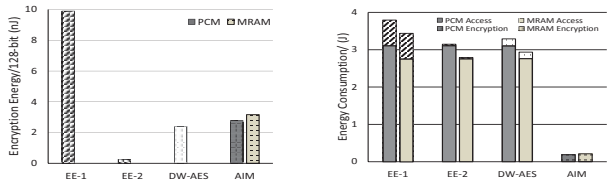

Figure 7: Left: Energy for encrypting 128-bit block. Right: Energy for accessing and encrypting 1GB main memory.

C. Overhead Evaluation

Figure 8 shows the area overhead. From this figure, **AIM** and **AIM-B** both incur insignificant area overhead of only 0.08% and 0.06% for MRAM-based main memory, and 0.6% and 0.4% area overhead for PCM-based main memory. Compared with AIM and AIM-B, AIM-S incurs a relatively larger area overhead of 3.6% and 5.0%.

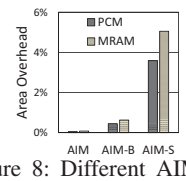

Figure 8: Different AIM designs area overhead.

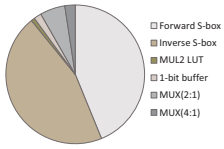

Figure 9: Breakdown of encryption overhead.

Figure 9 shows the distribution of hardware overhead. S-boxes have the largest area overhead. Since we can only look up byte by byte each time, more S-box LUTs mean higher parallelism. Thus, this overhead is unavoidable.

V. Conclusion

The proposed AIM encrypts many memory blocks simultaneously within each memory bank and the entire encryption process can be completed within the main memory without exposing the results to the memory bus. Experimental results show that AIM outperforms state-of-the-art AES engines with higher throughput yet lower energy consumption.

References

[1] "JEDEC DDR5 & NVDIMM-P Standards Under Development," https://www.jedec.org/news/pressreleases/jedec-ddr5-nvdimm-p-standards-under-development.

[2] "Intel: First 3D XPoint SSDs will feature up to 6GB/s of bandwidth," http://www.kitguru.net/components/memory/anton-shilov/intel-first-3d-xpoint-ssds-will-feature-up-to-6gbs-of-bandwidth, 2015.

[3] K. Chen *et al.*, "CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory," in *DATE*, 2012, pp. 33–38.

[4] S. Chhabra *et al.*, "i-NVMM: a secure non-volatile main memory system with incremental encryption," in *ISCA*, 2011, pp. 177–188.

[5] P. Colp *et al.*, "Protecting data on smartphones and tablets from memory attacks," in *ASPLOS*, 2015, pp. 177–189.

[6] X. Dong *et al.*, "NVSim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*, 2014, pp. 15–50.

[7] P. Hamalainen *et al.*, "Design and implementation of low-area and low-power AES encryption hardware core," in *DSD*, 2006, pp. 577–583.

[8] S. Li *et al.*, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *DAC*, 2016, pp. 1–6.

[9] T. Luo *et al.*, "A racetrack memory based in-memory booth multiplier for cryptography application," in *ASP-DAC*, 2016, pp. 286–291.

[10] 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, pp. 169–170.

[11] K. Tsuchida *et al.*, "A 64Mb MRAM with clamped-reference and adequate-reference schemes," in *ISSCC*, 2010, pp. 258–259.

[12] R. K. Venkatesan *et al.*, "Retention-aware placement in DRAM (RAPID): Software methods for quasi-non-volatile DRAM," in *HPCA*, 2006, pp. 155–165.

[13] Y. Wang *et al.*, "DW-AES: A Domain-Wall Nanowire-Based AES for High Throughput and Energy-Efficient Data Encryption in Non-Volatile Memory," *IFS*, vol. 11, no. 11, pp. 2426–2440, 2016.

[14] H.-S. P. Wong *et al.*, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[15] V. Young *et al.*, "DEUCE: Write-Efficient Encryption for Non-Volatile Memories," in *ASPLOS*, 2015, pp. 33–44.

[16] P. Zhou *et al.*, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009, pp. 14–23.