Securing Data Where It Makes Sense: In-Memory Encryption

Tolga Yalçın

Northern Arizona University, School of Informatics, Computing, and Cyber Systems, Flagstaff, AZ, USA.
e-mail: tolga.yalcin@nau.edu

ORCID ID: 0000-0001-7648-7669

Research Paper Received: 05.05.2020 Revised: 19.06.2020 Accepted: 21.06.2020

Abstract—Memory encryption has been an active research area in the recent decade. While the initial focus was on securing data in pervasive applications, recent efforts by Intel and AMD has brought memory encryption to general purpose processors as well. This has been mainly due to new threat models which necessitated securing real-time OS data inside RAM. The existing approaches use dedicated crypto engines that act as a buffer between the memory and the processor. In this study, we propose a novel approach where we combine a new paradigm in computing, in-memory processing, and cryptography to secure data inside the memory. We propose an in-memory encryption engine capable of utilizing processing capabilities of dynamic random access memories. We demonstrate the viability and efficiency of our proposal by implementing NSA cipher SIMON on our engine and show that encryption of a 1 Gb DRAM module can be completed in under 20 ms.

Keywords—In-memory processing, DDR, memory, encryption, SIMON.

1. Introduction

In a perfect world, there would be no need for securing data inside RAM, which is also known as *data in use*. The operating system should keep strong separation between processes and clear RAM upon reallocation to another process. In case the attack model allows analysis on the RAM, encrypting the swap area or using no swap would be sufficient. However, in real life applications, both operating systems and system administration are inherently imperfect human endeavors. It is always advisable to add some safeguards. Storing data in encrypted form in RAM is one of those safeguards, if not the most important one.

In practice, there can be several cases where memory can become visible to other processes, for example:

- Once the original process using the memory is completed, it returns the control to the operating system. Unless the memory is cleared, a successive process could perform a **malloc()** and retrieve info belonging to a previously running process [1].
- Pages swapped out to disk by the operating system can become available to a process watching the disk/storage [2].

It is possible to come up with several other attack scenarios where unencrypted memory can be probed by attackers. While, in general, this type of volatility depends greatly on the computer in question, it is safe to say that anonymous memory can persist for long periods of time, especially if the computer is idle. There has been several cases where passwords and other precalculated data were easily recovered from computers many days after being typed or loaded into memory [3].

A *cold boot attack* – also known as a *platform reset attack* [4],[5], for example, allows an attacker to perform a memory dump of a computer's random access memory (RAM) after power removal due to the phenomenon of computer data memory remanence. The attacker can retrieve encryption keys or other critical information from a running operating system.

Malicious hardware devices, rootkits and bootkits are other threats to memory [6]. They can directly infect one of the boot-start drivers inside the memory, thereby causing the operating system to load the modified driver upon start-up and causing the malicious code to take control.

Intel and AMD both introduced what is known as *Hardware-Assisted Trusted Execution Environment* to combat against such treats. Both solutions from the two major processor developers, namely *Intel Software Guard eXtensions* **(SGX)** [7] and *AMD Memory Encryption Technology* [8], rely on AES [9] based encryption engines implemented in hardware. In that sense, while they both offer proven-but-classical solutions which require major software changes and code refactoring (especially in case of Intel).

With the emergence of *In-Memory Processing* technology, in-memory encryption was discussed as a possible replacement or supplement for memory encryption. In [10] Seshadri et al proposed an in-memory accelerator for bulk bitwise operations, *Ambit*, and referred to memory encryption as one of the potential applications. However the authors did not further elaborate the idea.

In this study, we take where the authors left in [10], modify the Ambit accelerator to make it more suitable for memory encryption, and propose a conceptual in-memory encryption engine, which we refer to as *Secbit*. The proposed engine requires minimal modifications on the existing commodity DRAMs and memory controllers. It is also independent of encryption algorithm to be applied, as long as the target encryption algorithm relies on logical bitwise operations.

We then demonstrate viability of our engine by implementing NSA cipher SIMON [11]. We also present the performance results on a commodity 1 Gb DRAM.

The rest of this paper is organized as follows: In the next section, we summarize DRAM operation and architecture. It is followed by a description of our proposed in-memory encryption engine, *Secbit*. We then summarize our implementation of SIMON on *Secbit* together with performance figures. In our implementation, we use the 32/64 variant of SIMON, but also discuss extending our solution to other variants with performance implications. Finally, we discuss possible future directions and conclude our paper.

Our main contribution is the introduction of *Secbit* in-memory encryption engine, a modified version of Ambit, and demonstration of full memory encryption on *Secbit* by means of a internationally standardized encryption algorithm, SIMON, which has also been certified for use in *US national security systems for top secret communications*. To the best of our knowledge, this is the first study where a true in-memory encryption solution is proposed.

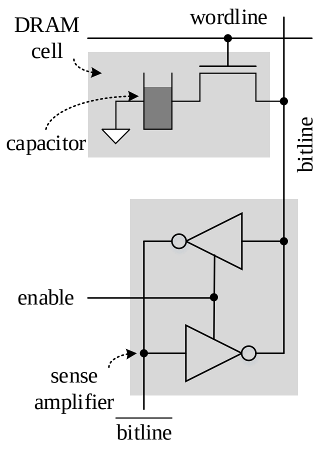

Fig. 1: DRAM cell and sense amplifier

2. DRAM Operation and Architecture

A random-access memory (RAM) that uses a single transistor-capacitor pair for each bit is called a dynamic random-access memory or DRAM [12]. Fig. 1 shows a single RAM cell together with the bit and word line, and the sense amplifier.

Data is stored inside the capacitor, where a fully charged capacitor represents logic-1 and a fully discharged capacitor represents logic-0. However, these capacitors are not perfect devices, and they leak charge. Therefore, they must be periodically refreshed (i.e., read and rewritten), making them dynamic in nature.

Furthermore, read and write operations involved in a DRAM cell are rather complicated compared to a static RAM. It has to go through several states in order for a read or write to be executed. Fig. 2 illustrates states involved during DRAM read operation.

These states can be summarized as follows:

1 Initially, the DRAM bitlines are in *precharged* state, i.e. kept at a constant voltage of $\frac{1}{2}V_{DD}$, and both wordline and bitline sense amplifiers are disabled.

2 For *read*, first, the wordline is enabled. This initiates a charge sharing between the DRAM capacitor and the bitline. Depending on the charge stored in the DRAM capacitor, the voltage on the bitline either increases (if capacitor was fully charged) or decreases (if capacitor was fully discharged) by a small amount, i.e. $\frac{1}{2}V_{DD} + \delta$ or $\frac{1}{2}V_{DD} - \delta$, respectively.

3 Then the sense amplifier is enabled. The sense amplifier is basically a pair of back-to-back connected inverters. Any change (either in the positive or negative direction) from the precharge value (which is also the equilibrium value for both inverters) will initiate an inversion operation on the inverter whose input is connected to the bitline. As its output goes low (or high), this will trigger the other inverter and cause its output to go high (or low) at an accelerated speed, thereby pulling the voltage on the bitline to either a full logic-1 or full logic-0. Once the inverters reach their steady state, not only will they have amplified the small change on the bitline to a full swing, but they will also fully charge or discharge the DRAM cell capacitor, whose wordline is still active.

4 The *write* operation will also follow a similar sequence. In that case, first the sense amplifiers will be enabled, transferring the full swing voltage sent to them for write to the bitline. The wordline will then be activated causing the target DRAM cell capacitor to be fully charged, or discharged.

5 Upon completion of a read or write sequence, both the wordline and sense amplifier will be disabled, and the bitline will return to its precharged state of $\frac{1}{2}V_{DD}$, ready for the next read or write.

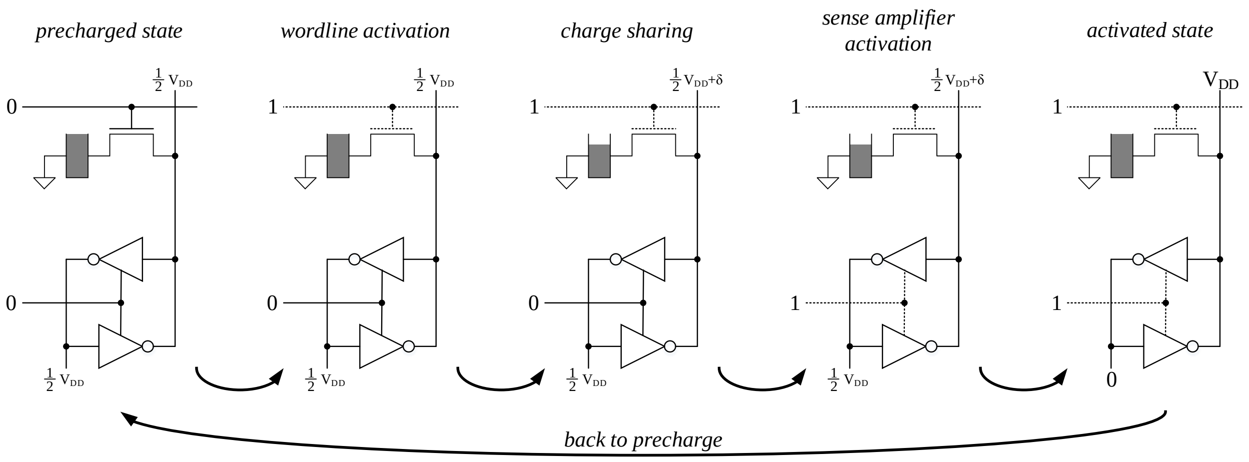

Fig. 2: DRAM read operation

A DRAM chip is a 2-dimensional array of these cells and sense amplifiers, organized in banks. For example, consider a 1 Gb DRAM with 8 banks. Each bank contains 128 Mb cells organized in 16384 rows (wordlines) and 8192 columns (bitlines). Wordline and bitline selection is done by the row and column decoders (see Fig. 3).

However, implementing large row and column decoders adversely affects the overall performance of DRAMs both in terms of speed and power consumption. Therefore, each bank is further divided into tiles (or mats), smaller arrays of 512×512 or 256×256 cells. Each group of tiles on the same row constitutes a subarray as shown in Fig. 4.

In addressing a row inside a bank, the memory controller splits the row address bits into two groups, one for selecting and precharging the subbank, the other for addressing the rows within all subarrays. This way, only the subarray bank which has the row to be read from or written to is activated, saving power. Column selection also occurs in a similar way. Columns only inside the target subarrays are selected and the outputs are sent to (or received from) the local row-buffer of the corresponding subarray. The global row-buffer communicates with the active row-buffer in order to transfer read or written data between the memory controller and the subarray.

Although, DRAM arrays have very wide widths and heights, their I/O bandwidth is rather limited to 8 or 16-bit words. In other words, from a 8192 cell column, we can only read 8 bits at a time. In order to read a full 64-bit word, 8 consecutive read operations have to be performed. The same applies to write operations as well. This, in fact, is one of the bottlenecks for encrypting DRAM data via external security engines. We refer the interested readers to [12] and [13] for further detailed information on DRAMs.

As we shall see in the next section, the suitability of DRAM physical structure for bitwise operations together with their subarray based hierarchical architecture will allow us to perform row operations with minimal delay inside the same subarray.

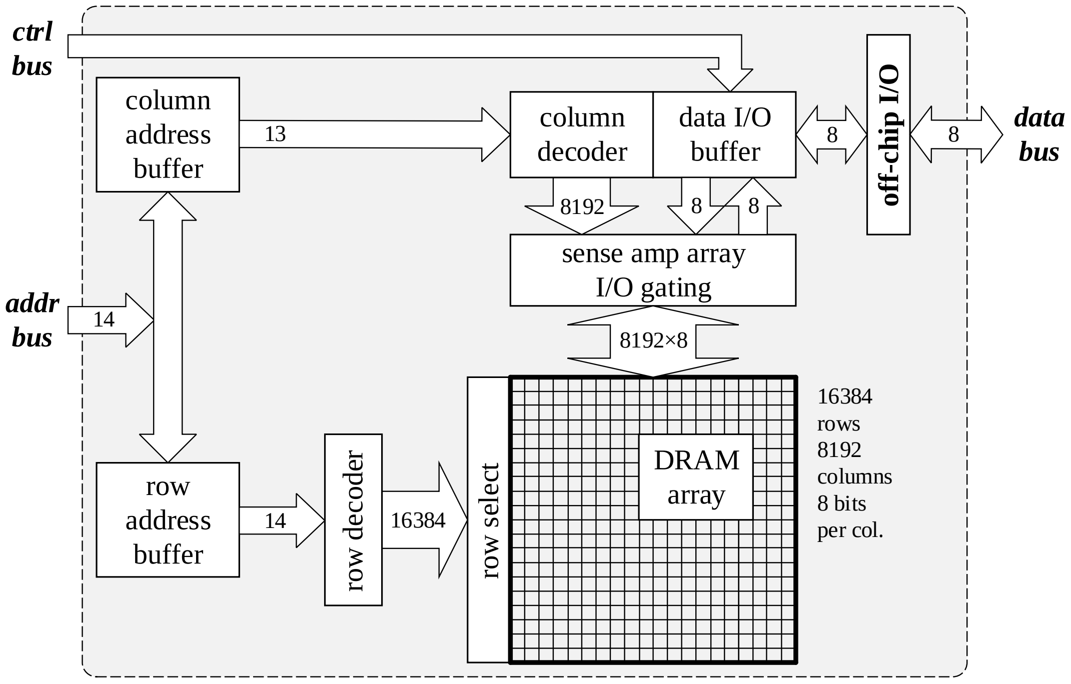

Fig. 3: DRAM architecture

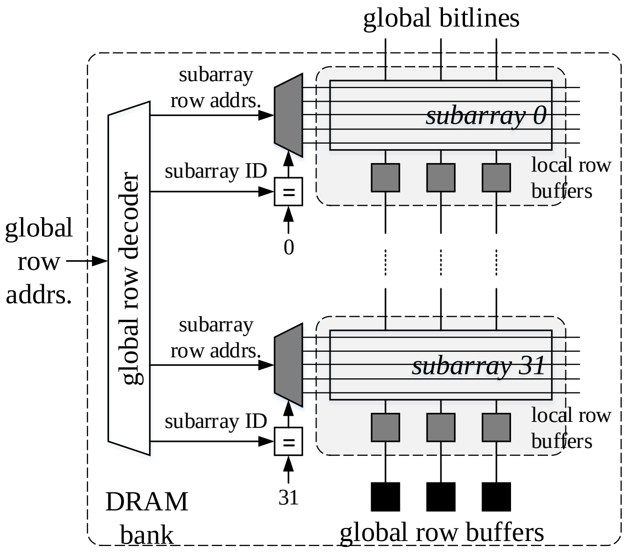

Fig. 4: DRAM bank physical organization

3. *Secbit* In-Memory Encryption Engine

In [10] and [14], authors proposed an in-memory accelerator capable of bitwise operations and fast transfers on row data inside DRAM memories, with minimal modifications on the memory array and the controller.

In this work, we modify this architecture for in-memory encryption operation. We propose to add five application-specific rows (ASR) to the subarray structure and modify the sense amplifier as shown in Fig. 5.

Let's briefly explain the functions of the additional rows:

- ***SR*** and ***TR*** (with wordlines SX and TX) act as source rows for bitwise operations. They can also be used for temporary data storage (similar to a general purpose in a processor).

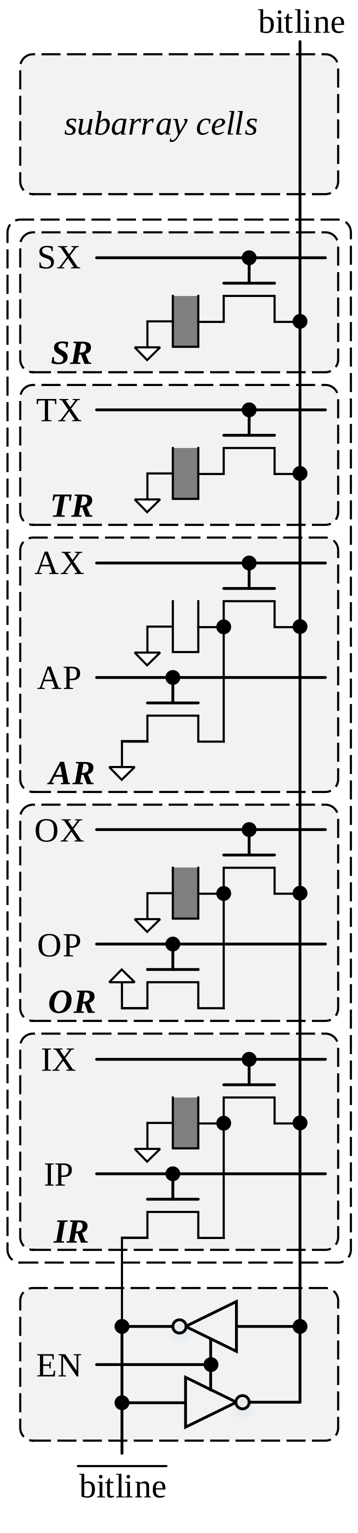

Fig. 5: *Secbit* rows

- ***aR*** is the AND row used for execution of AND operation. In addition to the wordline (AX) that connects the data capacitor to the bitline, it has a second wordline (AP) that connects the capacitor directly to GND line for precharging, which is essential for AND operation, as we shall see later.
- ***OR*** is the OR row used for execution of OR operation. Like the AND row, it has two wordlines, OX and OP for execution and precharging (to V_{DD}) for OR operation.
- ***IR*** is the INV row used for execution of inversion operation. In addition to the wordline (IX) to the bitline, its second wordline (IP) connects the capacitor to the $\overline{bitline}$ of the sense amplifier as in [10].
- The classical wordline structure of ***AR***, ***OR*** and ***IR*** further allow them to be used for temporary data storage, as in a general purpose register with special functionality.

The additional rows necessitate modification of the row decoder. However, as they are independent from the main array, the modifications are additive only, such as activation of more than one wordline, or activation of precharge wordlines, which have no effect on the main row decoder. Therefore, these additions can be embedded into the commands in the memory controller, which will be needed for memory encryption anyways.

Let's now see how the bitwise operations are executed on these additional registers.

3.1. Row Copy - RCP(WS,WD)

Although not a logical operation, row copy (RCP) is an essential operation for memory encryption. As we limit our focus on intra-subarray operations, we will be implementing row copy within the same subarray only, which is in fact the RowClone operation in [10].

It is accomplished by the following sequence of operations on any row inside the subarray, i.e. both regular rows (***RR***) and application-specific rows (***ASR***). Please note that, in the rest of this text we shall assume that all our arrays are initially in precharged state unless otherwise stated.

- Enable (activate) wordline of the source row (WS). This will change the voltage value on the bitlines.
- Activate sense amplifiers (SA). This will bring the bitlines to full logic values.
- Activate wordline of the destination row (WD). This will fully charge (or discharge) the capacitor of the cells on the destination rows to the same value as in the cells of the source rows.
- Disable wordlines and sense amplifiers and precharge the bitlines for the next operation. This will bring the subarray bitlines back to the initial value.

This scheme is illustrated in Fig. 6. The memory controller must be modified to allow consecutive activations of wordlines without precharge. The average timing for RCP is equal to $2t_{RAS} + t_{RP}$, which is less than 80 ns in modern DRAMs [15].

3.2. Row INV - RIV(WS)

Row inversion specifically uses the inversion row (IR) as destination for its special structure together with a source row. It is executed as follows:

- Activate wordline of the source row (WS). This will change the voltage value on the bitlines.
- Activate sense amplifiers (SA). This will bring the bitlines to full logic values.
- Activate precharge wordline of the ***IR*** (IP). This will fully charge (or discharge) the capacitor of the cells on ***IR*** to the inverse values of the cells of ***SR***.
- Disable wordlines and sense amplifiers and precharge the bitlines for the next operation. The inverted value is now inside ***IR***, ready for RCP.

The average timing for RIV is also less than 80 ns.

3.3. Row AND - RAN(S1,S2)

Row AND specifically uses the AND row (***AR***) for its special structure together with ***SR*** and ***TR*** as source rows. It destroys all data inside source rows, therefore data in the original source rows has to be copied to ***SR*** and ***TR*** prior to RAN. It is executed as follows:

- Copy data on the first source row (S1) to ***SR*** using RCP.
- Copy data on the second source row (S2) to ***TR*** using RCP. During the last activation step of RCP, also activate precharge wordline of ***ASR*** (AP). This will fully discharge ***AR*** capacitors.
- Activate wordlines of all three rows ***SR***, ***TR*** and ***AR***. This will initiate a charge-sharing on the bitlines in a manner similar to majority voting. Only if both capacitors on ***SR*** and ***TR*** are charged (logic-1), together with the zero on AR, this will bring the bitline above $\frac{1}{2}V_{DD}$, i.e. a logic-1 result. Even if either of the ***SR*** or ***TR*** capacitors is empty (logic-0), the average effect of the three capacitors on ***SR***, ***TR*** and ***AR*** will be to pull the bitline voltage below $\frac{1}{2}V_{DD}$, i.e. a logic-0 result. Hence the AND operation.
- Activate sense amplifiers (SA). This will bring the bitlines to full logic values. It will also fully charge or discharge all three rows ***SR***, ***TR*** and ***AR***.
- Disable wordlines and sense amplifiers and precharge the bitlines for the next operation. The AND value is now inside both ***SR***, ***TR*** and ***AR***, ready for RCP or another logic operation.

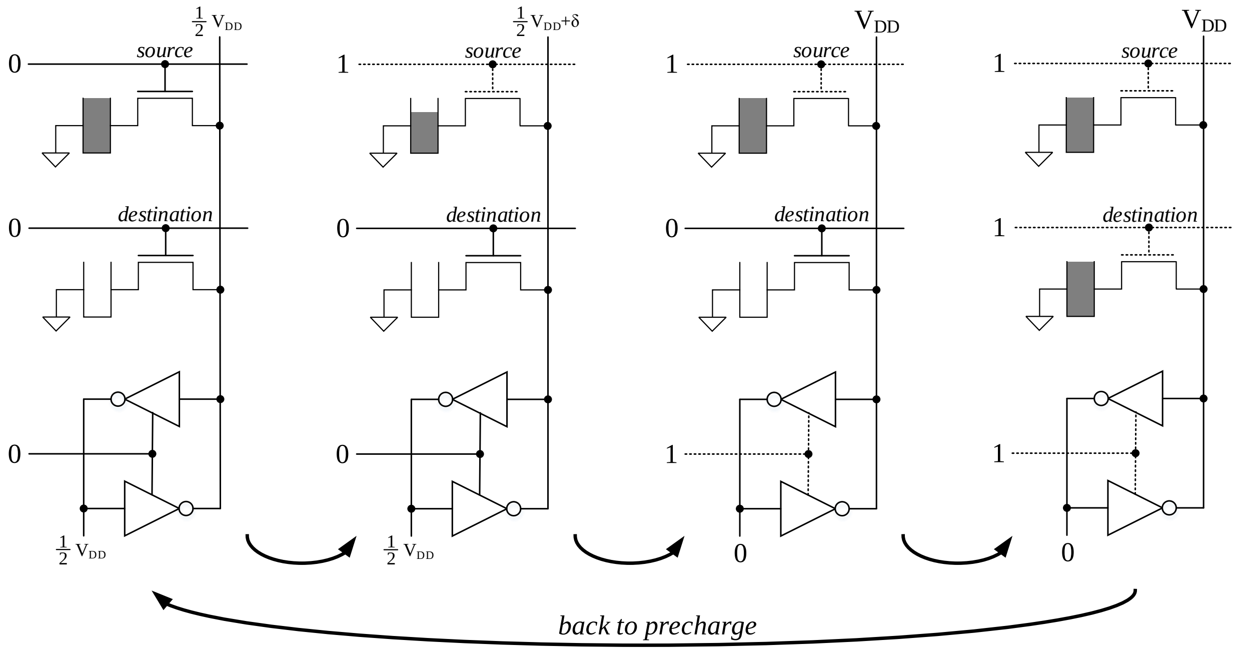

Fig. 6: RCP operation

Note that the memory controller has to be modified to allow simultaneous activation of the wordlines of three ***ASR***s, i.e. ***SR***, ***TR*** and ***AR***. The average timing for RAN is equal to the timing of two RCP and an AND operation (two activates and a precharge), which is less than 240 ns.

3.4. Row OR - ROR(S1,S2)

Row OR is almost identical to row AND, except OR row (***OR***) is used instead of AR and it is fully charged during the last activate steps in the second RCP by activating its precharge wordline (OP). At the end of the ROR sequence, the OR result is in all three registers ***SR***, ***TR*** and ***OR***, ready for RCP or another logic operation. Also, as in the case of RAN, memory controller has to be modified to allow simultaneous activation of the wordlines of these three ***ASR***s. The average timing for ROR is equal to that of RAN.

3.5. Row XOR - RXR(S1,S2)

There is no direct way of implementing Row XOR operation, RXR. Instead all the row operations above (RCP, RIV, RAN and ROR) are executed in the following sequence in order to compute RXR of data in two source rows, S1 and S2:

- Invert data on first source row (S1) using RIV. Then copy the result to ***SR*** using RCP.
- Copy data on second row (S2) to ***TR***, During the last activation step of RCP, also activate precharge wordline of ***AR*** (AP). This will fully discharge ***AR*** capacitors.
- Perform AND operation by activating wordlines of all three rows ***SR***, ***TR*** and ***AR***. Copy result to ***OR*** (used as temp register) using RCP.
- Invert data on second source row (S2) using RIV. Then copy the result to ***SR*** using RCP.
- Copy data on first row (S1) to ***TR***, During

the last activation step of RCP, also activate precharge wordline of ***AR*** (AP). This will fully discharge ***AR*** capacitors.

- Perform AND operation by activating wordlines of all three rows ***SR***, ***TR*** and ***AR***.
- Copy previous result from ***OR*** to ***TR*** using RCP. During the last activation step of RCP, also activate precharge wordline of ***OR*** (OP). This will fully charge ***OR*** capacitors.
- Perform OR operation by activating wordlines of all three rows ***SR***, ***TR*** and ***OR***. The XOR result is in all ***SR***, ***TR*** and ***OR*** rows ready for copy or a new operation.

The average timing for RXR is equal to the sum of timings of two RIV, two RAN and one ROR, which is less than 640 ns.

3.6. Row CLR - RCL(DS)

Row CLR clears the contents of a row by filling it with logic-0 (i.e. by discharging all data capacitors in that row) via ***AR***:

- Activate precharge wordline of ***AR*** (AP). This discharges its capacitors.
- Copy ***AR*** to destination row (DS) using RCP.

The average timing for RCL is equal to the sum of t_{RAS} and timing for RCP, which is less than 110 ns.

3.7. Row SET - RST(DS)

Row SET clears the contents of a row by filling it with logic-1 (i.e. by charging all data capacitors in that row) via ***OR***:

The average timing for RST is equal to that of RCL, i.e. 110 ns.

- Activate precharge wordline of ***OR*** (OP). This charges its capacitors.
- Copy ***OR*** to destination row (DS) using RCP.

4. SIMON on *Secbit*

In this section, we will summarize our implementation of NSA cipher SIMON on our in-memory encryption engine *Secbit*. Although of the two NSA ciphers, SPECK is proposed as the "software-friendly" and SIMON as the "hardware-friendly" candidate, this only holds for conventional processor architectures in which addition operation is supported by the arithmetic logic unit of the processor [16]. As explained in the text, the proposed *Secbit* engine has support for only basic logic functions such as Row Copy, Row INVert, Row AND and Row OR.

In other words, it allows programming using only four instructions. Even XOR operation needs to be implemented using a sequence of these four instructions. With its several binary additions, SPECK becomes an unsuitable choice for a *Secbit* implementation. This also applies to several Substitution-Permutation (SPN) based block ciphers, where both substitution (expressed as algebraic normal form) and permutation heavily rely on XOR operations.

On the other hand, SIMON requires only logical operations and hence is perfectly suited for *Secbit*. Let's start with a short reminder of how SIMON works before going into details of the implementation.

4.1. SIMON Cipher

SIMON is a family of lightweight block ciphers developed by the National Security Agency (NSA) [11]. It has been specifically optimized for performance in hardware implementations. It is a balanced Feistel cipher with an n-bit word, and therefore block length of $2n$, whereas the key length is a multiple of n by 2, 3, or 4, which is the value m. Therefore, a SIMON variant is denoted as SIMON-$2n/nm$. For example, SIMON-32/64 refers to the

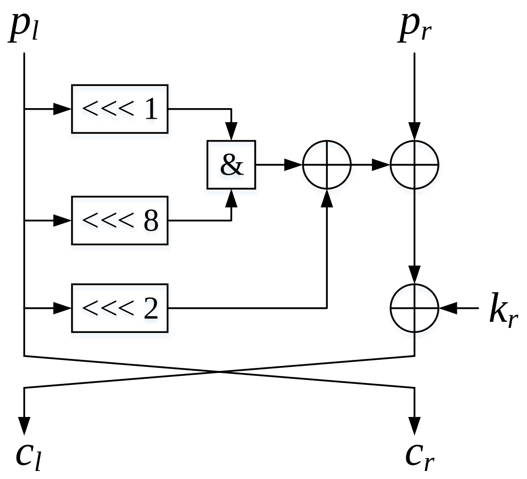

Fig. 7: SIMON round

cipher with a 32-bit plaintext block ($n = 16$) and a 64-bit key. The round operation of the cipher is uniform between variants, while number of rounds depend on n. Similarly, the key generation depends on the value of m. Fig. 7 shows one round of SIMON.

4.2. Bit-Sliced SIMON

As our subarray row operations do not allow shift or rotate operations used in SIMON, we cannot implement it on rows directly. Therefore, although SIMON is a cipher optimized for hardware, we will treat it as a cipher optimized for software and implement it in a bit-sliced fashion. Not unlike conventional implementations, where the wordlength of bit-sliced implementation is limited to the width of processor registers (words), we take our DRAM rows as words and do our coding accordingly.

For our demonstration, we choose the smallest variant of SIMON, i.e SIMON-32/64, with a total of 32 rounds. This corresponds to a block size of 32 bits, whereas in our implementation it corresponds to a DRAM block of 32 rows. Furthermore, we leave the key generation out of the scope of this implementation. For key generation, we propose the following solutions:

- The key schedule can be pre-executed by the main controller and sent to memory controller, which can then turn it to a bit-sliced format and store inside a predetermined location of the memory. This solution will possibly be problematic as the key is stored in un-encrypted inside the memory leaving it open to attacks.
- The memory, upon power-up, can be used as a physically unclonable function (PUF) and generate its own encryption key. In this case, key schedule does not need to be executed. All the round keys can be extracted from PUF, and stored like a one-time-pad inside the memory. This scheme can in fact be applied on each bank separately introducing additional security. This will require major modifications on the firmware (PUF initialization upon power-up) and is likely to cause additional security problems. For once, the key or the keys extracted from the PUF will be stored un-encrypted inside the memory. One solution may be application of a simplified key encryption scheme by the memory controller.
- The key can be stored inside *Secbit* engine, which will be an integral part of the memory controller. This way, *Secbit* can incorporate the key into its program execution. There may be side-channel issues related to the execution of the program. However, counter-measures for such attacks are well-documented and fairly easy-to-apply [17].

In our bit-sliced implementation, a 32-bit SIMON block corresponds to 32 rows inside the subarray, organized as 16 rows for each of the left and right words. We will refer to each or these rows as R_i

where $i \in [0, 31]$. In each subarray, we will dedicate 16 rows for temp data storage and refer to them as Q_i where $i \in [0, 15]$.

One round of SIMON-32/64 can be written as

$$\begin{aligned} c_l &= (\,(p_l <<< 1) \,\&\, (p_l <<< 8)\,) \\ &\quad \oplus\ (p_l <<< 2) \oplus p_r \oplus k_r \\ c_r &= p_l \end{aligned}$$

For our SIMON-32/64 variant, we can express the first equation in bitwise form as:

$$\begin{array}{c} c_l\,[0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\,] \\ = \\ (\,p_l\,[1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 0\,] \\ \& \\ p_l\,[8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\,]\,) \\ \oplus \\ p_l\,[2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 0\ 1\,] \\ \oplus \\ p_r\,[0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\,] \\ \oplus \\ k_r\,[0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\,] \end{array}$$

```
**for** $round = 0$ to 15 **do**
    **for** $i = 0$ to 15 **do**
        $T_i \leftarrow (\, R_{i+1,16} \,\&\, R_{i+8,16} \,)$
            $\oplus\ R_{i+2,16} \oplus R_{i+16} \oplus K_{32 \cdot round + i}$
    **end for**
    **for** $i = 0$ to 15 **do**
        $R_{i+1,16} \leftarrow T_i$
    **end for**
    **for** $i = 0$ to 15 **do**
        $T_i \leftarrow (\, R_{i+1,17} \,\&\, R_{i+24,32} \,)$
            $\oplus\ R_{i+18,16} \oplus R_i \oplus K_{32 \cdot round + 16 + i}$
    **end for**
    **for** $i = 0$ to 15 **do**
        $R_{i,16} \leftarrow T_i$
    **end for**
**end for**
```

Fig. 8: Pseudo-code for one round of SIMON-32/64

We convert this to a row-based pseudo-code given in Fig. 8, where the first two internal loops process rows R_0 to R_{15} as left plaintext word and rows R_{16} to R_{31} as right plaintext. The result written to temp rows T_0 to T_{15} are then copied back to rows R_{16} to R_{31}. In the second two internal loops the roles of the loops interchange and processing is done accordingly. At the end of the iteration of all internal loops, two rounds of encryption is completed, and left and right plaintext words are back where there belong. The outer loop is executed 16 times for 32 rounds.

In actual execution, each update of T_i is much more complicated. The pseudo-code in Fig 9. shows the execution steps for computation of T_0 in the first loop:

In this pseudo-code, we assume that the key bit (stored in a full row) is set in advance to either logic-0 or logic-1 via RCL or RST, respectively. This way, the execution time of the loop is constant as a counter-measure against timing attacks. In a timing optimized code the RXR with key can be replaced by RIV (if key bit is logic-1) or simply skipped (if key bit is logic-0).

The code stream above only computes 1-bit of the 16-bit left ciphertext. It has to be repeated 16 times for each round, and then 32 times for 32 rounds.

```
**RAN**( $R_1$ , $R_8$ )     // $AR \leftarrow R_1 \wedge R_8$
**RCP**( $AR$, $T_0$ )     // $T_0 \leftarrow AR$
**RXR**( $T_0$ , $R_2$ )     // $OR \leftarrow T_0 \oplus R_2$
**RCP**( $OR$, $T_0$ )     // $T_0 \leftarrow OR$
**RXR**( $T_0$ , $R_{16}$ )    // $OR \leftarrow T_0 \oplus R_{16}$
**RCP**( $OR$, $T_0$ )     // $T_0 \leftarrow OR$
**RXR**( $T_0$ , $K_0$ )     // $OR \leftarrow T_0 \oplus K_0$
**RCP**( $OR$, $T_0$ )     // $T_0 \leftarrow OR$
```

Fig. 9: Pseudo-code for computation of T_0 in the first loop

5. Performance Results

Each row computation requires one RAN, three RXR and four RCP, resulting in a total execution time of $240 + 3 \times 640 + 4 \times 80 = 2480$ ns. In each round, this row computation is repeated 16 times, followed by 16 row copies (RCP), resulting in a total of $16 \times 2480 + 16 \times 80 = 40960$ ns. Repeated 32 rounds, this results in 1.31 ms of encryption time for 32 rows of data. In our 1 Gb DRAM example, there exist a total number of 16384 rows organized in 8 banks. This means that full memory encryption will take $(16384/32) \times 8 \times 1.31 = 5.37$ s, which corresponds to 5 ns/bit encryption time. Decryption time will be equal to encryption due to the Feistel structure of SIMON.

While internal architecture of DRAM chips do not allow processing on different subarrays within a bank, parallel processing in different banks is possible. In an 8-bank DRAM, this will reduce the encryption time to 0.625 ns/bit. This corresponds to 20 ns per 32-bit blocks. With a custom hardware implementation of SIMON-32/64 as an off-chip encryption engine, this would correspond to each clock cycle being completed in $20/32 = 0.625$ ns, i.e. 1600 MHz operation. This is based on the assumption that there would be no additional DRAM read-write delays.

On the other hand, using an AES-128 based encryption engine would be more favorable. It would be able to process 128-bit blocks in 10 clock cycles, corresponding to 80 ns which *Secbit* requires for 128-bit encryption. That is 8 ns per clock cycle, i.e. 125 MHz operation – again without any additional RAM read-write delays taken into account. It should also be noted that while SIMON-32/64 engine would require less than 700 GE for ASIC implementation, a high performance parallel AES-128 would require above 13K GE [18], whereas *Secbit* implementation requires no external hardware.

It is also worth mentioning other state-of-the-art work reported in [19]-[22], even though none of them target DRAMs as the memory platforms, making them unsuitable for comparison with our proposed engine and performance figures.

- AES in-memory (AIM) implementation in [19] targets emerging non-volatile memory (NVM) technologies such as resistance-based storage and current sensing. It utilizes certain features specific to NVMs, such as ability to implement direct XOR operation, and can realize AES algorithm rather effectively.
- PIMA-Logic in [20] is implemented for Spin Orbit Torque Magnetic Random Access Memory (SOT-MRAM) array. It can simultaneously work as a non-volatile memory and a reconfigurable in-memory logic, making it very suitable for AES encryption.
- MRIMA in [21] is an MRAM-based in-memory accelerator. It is specifically designed with convolutional neural networks (CNNs) in mind. Still, it can implement AES encryption much more effectively compared to an off-memory implementation.

- In-memory computing architecture in [22] based magneto-electric random access memory (MeRAM), where precessional magnetism of MeRAM is utilized to carry out XOR encryption of the device state with a key. However, it does not present a full encryption solution (AES or otherwise).

Another important point needed to be discussed is the significance of 20 ms required for full memory encryption: In a real-time operation sysmtem (RTOS), memory allocation-release times vary from 1 to 10 μs, depending on the process and memory block size. The new memory manager ERMM in [23] requires 1.036 and 0.986 μs for allocation and release of 128 byte memory blocks, respectively. With our in-memory encryption scheme, 1600 bits (200 bytes) can be encrypted/decrypted in 1 μs, which more than matches these figures. We expect our in-memory encryption scheme to have little to no effect on RTOS performance.

6. Conclusion and Future Work

In this study, we have introduced an in-memory encryption engine, *Secbit*, suitable for DRAMs and demonstrated its efficiency by means of a proven cipher, NSA's SIMON. Our engine requires minimal modifications on the existing DDR architectures with an area overhead of $< 1\%$. Using the smallest variant of SIMON, it can encrypt 2 Mb of data inside 8 banks of a 1 Gb DDR in less than 1.31 ms at a cost of 3.125 % unusable memory (16 rows in every 512 reserved for temporary data storage). Full memory can be encrypted in less than 20 ms. For additional security, other variants of SIMON can be used, at the cost of additional unusable memory and higher encryption times. For example, going with SIMON-64 will double both the rows reserved for temporary data and encryption time.

While these figures are encouraging, they are far from being optimal. It is possible to further minimize these numbers by carefully manipulating the timings on the DDR row decoders. Several activations can be overlapped. Furthermore, the timings figures presented in this study refer to DDR3 RAMs. As the RAM technology is improved, they also get lower. In a DDR4 RAM, these figures will lower by at least 20 %. It is also possible to reduce timings by carefully re-ordering the executing of instructions.

Moreover, we used a standard cipher here, one which is in fact optimized for hardware implementation. With *Secbit* in mind, it is possible to design ciphers optimized for in-memory encryption, in a way similar to ciphers optimized for software implementation [24]. We leave this as an open research problem.

References

[1] A. Rubini and J. Corbet. Linux device drivers (nutshell handbooks), 1998.

[2] D. Gruss, et al. Page cache attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 167–180, 2019.

[3] D. Farmer and W. Venema. *Forensic discovery*. Addison-Wesley Professional, 2009.

[4] J. A. Halderman, et al. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[5] R. Carbone, et al. An in-depth analysis of the cold boot attack. *DRDC Valcartier, Defence Research and Development, Canada, Tech. Rep*, 2011.

[6] A. Matrosov, E. Rodionov and S. Bratus. *Rootkits and bootkits: reversing modern malware and next generation threats*. No Starch Press, 2019.

[7] F. McKeen, et. al. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.

[8] D. Kaplan. {AMD} x86 memory encryption technologies. 2016.

[9] V. Rijmen and J. Daemen. Advanced encryption standard. *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pages 19–22, 2001.

[10] V. Seshadri, et al. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287. IEEE, 2017.

[11] R. Beaulieu, et al.and Louis Wingers. The simon and speck lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[12] B. Jacob, D. Wang and S. Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[13] K. Itoh. *VLSI memory chip design*, volume 5. Springer Science & Business Media, 2013.

[14] V. Seshadri. Simple dram and virtual memory abstractions to enable highly efficient memory systems. *arXiv preprint arXiv:1605.06483*, 2016.

[15] V. Seshadri, et al. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197, 2013.

[16] J. Hennessy and D. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] S. Mangard, E. Oswald and T. Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[18] N. Pramstaller, S. Mangard, S. Dominikus and J. Wolkerstorfer. Efficient aes implementations on asics and fpgas. In *International Conference on Advanced Encryption Standard*, pages 98–112. Springer, 2004.

[19] M. Xie, et al. Securing emerging nonvolatile main memory with fast and energy-efficient aes in-memory implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(11):2443–2455, 2018.

[20] S. Angizi, Z. He and D. Fan. Pima-logic: a novel processing-in-memory architecture for highly flexible and energy-efficient logic computation. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[21] S. Angizi, Z. He, A. Awad and D. Fan. Mrima: an mram-based in-memory accelerator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[22] A. Lee and K.-L. Wang. Full memory encryption with magneto-electric in-memory computing. In *2019 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)*, pages 1–2. IEEE, 2019.

[23] X. Huang. Construction on embedded real-time operating system of computer. In *2015 2nd International Conference on Electrical, Computer Engineering and Electronics*. Atlantis Press, 2015.

[24] M. R. Albrecht, et al. Block ciphers–focus on the linear layer (feat. pride). In *Annual Cryptology Conference*, pages 57–76. Springer, 2014.