2020

Enabling Recovery of Secure Non-Volatile Memories

Mao Ye
University of Central Florida

ENABLING RECOVERY OF SECURE NON-VOLATILE MEMORIES

by

MAO YE
B.S. Zhejiang University, 2004
M.S. University of Central Florida, 2017

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2020

Major Professor: Amro Awad

ABSTRACT

Emerging non-volatile memories (NVMs), such as phase change memory (PCM), spin-transfer torque RAM (STT-RAM) and resistive RAM (ReRAM), have dual memory-storage characteristics and, therefore, are strong candidates to replace or augment current DRAM and secondary storage devices. The newly released Intel 3D XPoint persistent memory and Optane SSD series have shown promising features. However, when these new devices are exposed to events such as power loss, many issues arise when data recovery is expected. In this dissertation, I devised multiple schemes to enable secure data recovery for emerging NVM technologies when memory encryption is used. With the data-remanence feature of NVMs, physical attacks become easier; hence, emerging NVMs are typically paired with encryption. In particular, counter-mode encryption is commonly used due to its performance and security advantages over other schemes (e.g., electronic codebook encryption). However, enabling data recovery in power failure events requires the recovery of security metadata associated with data blocks. Naively writing security metadata updates along with data for each operation can further exacerbate the write endurance problem of NVMs as they have limited write endurance and very slow write operations. Therefore, it is necessary to enable the recovery of data and security metadata (encryption counters) but without incurring a significant number of writes.

The first work of this dissertation presents an explanation of Osiris, a novel mechanism that repurposes error correcting code (ECC) co-located with data to enable recovery of encryption counters by additionally serving as a sanity-check for encryption counters used. Thus, by using a stop-loss mechanism with a limited number of trials, ECC can be used to identify which encryption counter that was used most recently to encrypt the data and, hence, allow correct decryption and recovery. The first work of this dissertation explores how different stop-loss parameters along with optimizations of Osiris can potentially reduce the number of writes. Overall, Osiris enables

the recovery of encryption counters while achieving better performance and fewer writes than a conventional write-back caching scheme of encryption counters, which lacks the ability to recover encryption counters. Later, in the second work, Osiris implementation is expanded to work with different counter-mode memory encryption schemes, where we use an epoch-based approach to periodically persist updated counters. Later, when a crash occurs, we can recover counters through test-and-verification to identify the correct counter within the size of an epoch for counter recovery. Our proposed scheme, Osiris-Global, incurs minimal performance overheads and write overheads in enabling the recovery of encryption counters.

In summary, the findings of the present PhD work enable the recovery of secure NVM systems and, hence, allows persistent applications to leverage the persistency features of NVMs. Meanwhile, it also minimizes the number of writes required in meeting this crash consistency requirement of secure NVM systems.

This dissertation is dedicated to my parents, Weiyin Ye and Lining Wu, the kindest persons in my life who have given me unconditional support for my dreams.

ACKNOWLEDGMENTS

I would like to express my gratitude to my adviser, Prof. Amro Awad, for his patience and guidance on my research work, his instructions on my writing and presentation and his tremendous efforts on my summer internship search. Without his help, It would have been impossible for me to publish at top-tier conferences, meet many smart and interesting students and scholars in computer architecture research community and finish this dissertation.

Additionally, Dr. Mingjie Lin and Dr. Shaojie Zhang provide valuable suggestions to me on many occasions. Dr. Rickard Ewetz always gives me frank and sharp opinions and the discussion with Dr. Fan Yao during joint seminar is informative. Further, Dr. John Seel gives me incredible encouragement and support. I am very glad to have them as my committee members and herein thank them sincerely. Moreover, I want to thank Dr. Kalpathy Sundaram, Dr. Yan Solihin, Dr. Aziz Mohaisen and Dr. Clayton Hughes respectively, for either giving me personal suggestions or instructions on publications. My appreciation also goes to Diana Poulalion for her kindness and honesty.

Furthermore, I would like to express my appreciation to my labmates: Vamsee Reddy, Mazen AlWadi, Kazi Zubair, John McFarland, Ghadeer Al-Musaddar and Abdullahal Arafat. We shared short but precious good times together that I will remember forever. My special thanks go to Nicholas Omusi, who has given me his emotional support from beginning to end.

In addition, I would like to thank some of my friends in academia and life. Beginning with my friends in academia first, they are Youwei Zhuo, Sihang Liu, Linghao Song, Pengfei Zuo, Xin Hu, Haiyu Mao, Tianqi Tang, Yifan Sun and Mengjia Yan. I met them during conferences, have had interesting and useful discussion with them and have received their valuable support. From my personal life, I would sincerely like to thank, Jie Xiong, who treated me to her delicious Sichuan

cuisine many times when I was too busy with deadlines and provided me valuable personal advice during my down time. I also owe a lot to Xinyan Zha, my life-long friend, who keeps calling me every week for months to help me get through the difficult times, and to Huihui Li, who offered me several referrals and, sent me preparation materials and useful information. I would also like to deeply thank Yu Bi and Kate Wang. With their sincere help, I received my current job. Besides, I would like to thank Xuhong Zhang, Jun Ye, Jiangling Yin, Dan Huang, Hao Hu, Bin Huang, Yuyan Bao, Sudharsan Vaidhun, Dean Sullivan, Dr. Zakhia Abichar, Dr. Suboh Suboh, Dr. Deliang Fan, Professor Su-I Hou, Paster Chen, Luo Yu, Biquan Wang, Lin He, Xin Liu, Tia Law and many others for their kindness. Again, I want to thank Prof. Hongwei Yang for his encouragement and help.

Lastly, I would like to thank my extended family in China and in US for their support before and during the pandemic and through the whole PhD journey.

TABLE OF CONTENTS

LIST OF FIGURES xi

LIST OF TABLES xiii

CHAPTER 1: INTRODUCTION AND BACKGROUND 1

Memory Encryption 1

Non Volatile Memories 2

Recovery for Secure Non-volatile Memory 7

CHAPTER 2: OSIRIS 11

Background and Motivation 12

Background 12

Emerging NVMs 12

Memory Encryption and Data Integrity Verification 13

Encryption Metadata Crash Consistency 16

Design 18

Threat Model 18

Design Assumption 19
Design Options 19
Design 22
Reliability and Recovery from Crash 29
Different ECC Schemes 32
Systems without ECC 34
System with both ECC and MAC 35
Security of Osiris and Osiris-Plus 35
Evaluation 36
Methodology 36
Analysis 37
Impact of Osiris Limit 38
Impact of Osiris and Osiris Plus persistency on multiple benchmarks 40
CHAPTER 3: OSIRIS GLOBAL 43
Background 43
Design 44
Write Operation of Osiris-global 47

Hardware Overhead for Osiris-global 49

Evaluation 50

The impact of Epoch Number 50

Osiris-global persistency on multiple benchmarks 51

Recovery Time 53

CHAPTER 4: SUMMARY 55

LIST OF REFERENCES 57

LIST OF FIGURES

1.1 Counter-mode encryption in memory 8

1.2 Demo construction of Merkle Tree and Bonsai Merkle Tree 9

2.1 State-of-the-art counter mode encryption. AES is shown but other cryptographic algorithms are possible. 13

2.2 An example Merkle Tree for integrity verification. 14

2.3 Steps for write operations to ensure crash consistency. 16

2.4 Osiris write operation 27

2.5 Osiris read operation 27

2.6 Osiris-Plus read operation 29

2.7 The impact of Osiris limit on performance. 38

2.8 The impact of Osiris limit on number of writes. 39

2.9 The impact of Osiris and Osiris Plus persistence on performance. 41

2.10 The impact of Osiris and Osiris Plus persistence on number of writes. 42

3.1 The structure of the Epoch Record Table (ERT) and the counter cache 46

3.2 The flowchart of Epoch Record Table (ERT) update and counter cache update 49

3.3 Osiris-global write operation 50

3.4 The impact of Epoch Number on performance (Number of writes). 51

3.5 The sensitivity studies of cache size for Osiris-global 52

3.6 The impact of Osiris-global on performance. 53

3.7 The impact of Osiris-global on number of writes. 54

LIST OF TABLES

2.1 Common sources of ECC mismatch 24

2.2 Configuration of the simulated system. 36

3.1 Counter Types Used in Encryption Mode 44

CHAPTER 1: INTRODUCTION AND BACKGROUND

This chapter provides the general information regarding the current progress of emerging non-volatile memory(NVM) technologies with an emphasis on memory encryption and secure metadata restoration.

Memory Encryption

Memory encryption is an important measure to protect memory content from being modified or leaked out intentionally or unintentionally. Following the lead on storage and network device encryption, memory encryption that started as a feature for specific products has now become a common feature for general products. Memory encryption has drawn attention and is widely employed because memory can be attacked easily when industries rapidly move their business and critical data to cloud platforms where computation and storage resource is shared. Recent years, more and more hardware vulnerabilities are probed and revealed from security research community to the public[1]. The most frequently seen attacks include but are not limited to bus-snooping attack, memory scan attack(i.e. cold boot) and, memory content tampering attacks (i.e. memory spoofing, splicing and packet/data replay)[2]. At the front line of the battle for guaranteeing security, AMD proposes secure memory encryption (SME) to provide full memory encryption for their products. With SME, each memory controller is equipped with a standard AES engine for data encryption/decryption. The AES engine uses a key that is generated on each reset and stored on a system-on-chip register only visible to hardware to guarantee its security[1]. Intel, on the other hand, proposes a technology called Software Guard Extension to mitigate security risks that arise from using shared resources and untrusted operating system or untrusted DRAM[3, 4]. The core component of this technology is Memory Encryption Engine (MEE). While the system is booting

up, it reserves a fixed portion of memory as a data protection region where all the data will be encrypted. Therefore, if an application requires high-level of security during run-time, it will be only mapped to the encrypted memory. Additionally, data integrity is supported as a part of MEE unit[3]. SGX, although powerful, cannot guarantee the security for the whole memory. Therefore, Intel still expects to implement full-memory encryption in the future, similar to current AMD's SME technology[5].

While encrypted DRAM has become more developed, encrypted non-volatile memory (NVM) is new to the market[6, 7]. Due to its non-volatility NVM is more vulnerable than DRAM for data leakage attack. Despite removal from the device or under power shortage, NVM keeps its data intact. Moreover, all the aforementioned memory attacks apply to NVM as well. In particular, data replay attack is a threat model of the interests of this dissertation. Hence, encryption is required for NVM in concern of security risk. To make NVM feasible for current heterogeneous environment and shared cloud platform, NVM encryption and its consequence(i.e. performance, trade off, consistency) need open discussion and careful examination.

Non Volatile Memories

Emerging NVMs, as the latest products to enter the memory market, demonstrate promising characteristics for users. NVMs such as phase change memory (PCM)[8], resistive RAM (ReRAM)[9], spin-transfer torque RAM (STT-RAM)[10] show better scalability, higher density and cheaper per-bit costs[11].

The major difference between these NVMs stems from the materials and the structures on which they are built[8, 9, 10]. PCM is temperature-dependent technology to make chalchogenide alloy change between amorphous and crystalline phases. The amorphous phase results from a low

temperature and shows a high resistance whereas crystalline phase arrives only when the temperature is above the crystallization point. The different resistance level marks the different stored value[12]. A cell of ReRAM has a sandwich structure made of metal/oxide insular/metal layers. Similarly, it can also have two distinguished resistance levels due to the filament formation to enable conductive path in the insular layer[13]. By adjusting the voltage through an external source, the low-resistance and high-resistance statuses can be efficiently switched, representing logical 0 and 1[9]. STT-RAM cells are also built with a sandwich structure as well, with two ferromagnetic layers and one tunnel barrier layer in between. When the magnetic directions of two ferromagnetic layers are the same, it represents logical 0 and, vice versa, it represents logical 1[14, 10]. Therefore, it is the physical characteristics of the materials that determine the performance pattern of NVMs. Even within the same category, the variance can be huge[15, 8]. This dissertation focuses on phase-change memory, which is considered the core technology for both the Intel Optane Persistent Memory and Intel Optane SSD Series[7]. Phase-change memory has clear advantages, such as DRAM-comparable processing speed (read only), high capacity, byte addressability and non-volatility[8]. Overall, NVM exhibits both memory and storage characteristics. Since 2009, NVM technology has been commonly seen in NAND flash and SSD[16], and, more recently, in hybrid or unified filesystems as the main storage powered by advanced 3D XPoint technology, especially in data centers[17, 6]. With PCM serving as the main storage of the file system, new applications can unleash their performance by taking advantage of the byte addressability of PCM through direct load/store operations[18, 19]. Hence, using PCM can yield tremendous performance gains, but it requires changes for most legacy applications[20, 21].

On the other hand, PCM has drawbacks. It is known for limited write endurance, slow writes compared to DRAM, power-consuming writes, and, finally the non-volatility, which appears as a weak point in some scenarios due to the data-remanence vulnerability. To address these drawbacks, prior studies have intensively researched in these challenges [22, 23, 24, 25, 26, 27, 28, 29, 30].

Their major results and solutions are summarized below.

It is shown that the write limit of PCM is around 10^7 to 10^8 with current technology[23]. This is because write operations require frequent heating and cooling to memory cells and, to a threshold, a cell ends up permanently losing its programmability[31]. While it is not plausible to limit the number of writes that each application needs to complete its task, it is useful to reduce unnecessary writes or distribute writes evenly to the cells on the memory. As such, multiple methods are addressed. In flip-and-write, the authors propose a simple read-modify-write solution to avoid unnecessary bit programming by only modifying changed bits. In addition, they introduce a flip bit which is set when the read comparison finds more than half of the bits between the old data and new data are different. Then, the old data chunk flips all of its bits to the opposite value before writing the modified part of the new data to the memory[32]. These two mechanisms can reduce write programming by half. To balance writes onto different locations of the memory, the write count for each memory line has to be tracked and the relocated line needs to be calculated. The Start-Gap wear-leveling algorithm periodically moves the content of a memory line to the line next to it, regardless of the write traffic[23]. To achieve this goal, it requires two registers named Gap and Start respectively and, one extra line to facilitate data movement. Then, for every N writes, starting from the line adjacent to the gap line, its content is copied to the gap line, making itself become the next gap line. Gap register always points to the gap line and, when Gap reaches 0, the memory works like a circular buffer to copy the content of the gap line at the beginning to the line that the Start register points to. To start a new data movement cycle, the Gap register again is set to N+1, pointing to the gap line and Start register increments to record the write cycle number. However, this method does not help with the cases where writes are frequently positioned to adjacent memory lines. To optimize this algorithm, a randomness generation function is used to enhance the location randomness of the gap line. Overall, the Start-Gap method can achieve up to 97% of the ideal device lifetime [23] after the optimization. Similarly, Security Refresh

proposes to use a randomized address mapping mechanism to hide the memory locations from users and software [25]. In this study, physical locations of the data are constantly migrated within the memory. They can elongate the lifetime of PCM for another six years when confronted with the worst adversary attack described in their paper to exacerbate the write endurance to a specific set of cells[25]. The type of adversary attack to wear out cells by repeated writes is also addressed in the Start-Gap paper[23]. Their solution is to partition the memory into different sections and apply Start-Gap wear-leveling to each section. This is to reduce the number of writes to finish a cycle of data swapping for each partition. With this region-based Start-Gap, they can increase PCM's life by months to even years under the described attacks in their paper [23].

For PCM that is paired with counter-mode encryption, a number of studies have proposed different solutions to enhance device endurance, reduce write power and increase write bandwidth[22, 24]. Before discussing these works in details, one needs to understand that data remanence vulnerability of PCM is obvious and lasts much longer than DRAM. Needless to say, it has no temperature constrains like DRAM if one attempts to retrieve data from memory devices with no power supplies.

In DEUCE, it shows that with current counter-mode encryption, when the actual number of different bits between a chunk of old and new plaintext is small, the encrypted text written to the PCM could end up with around a 50% bit difference. Considering the fact that most written-back data only has a difference of few words from what is present in the memory, this work proposes to only re-encrypt the part that has been changed on each cacheline. In order to do so, this paper tracks the cacheline difference, and provides a two-counter system to reduce unnecessary bit flip. The two line counters are based on the physical cacheline counter. One is the same as the physical counter and the other masks the unchanged part of the cacheline bounded by an Epoch Interval. Using this scheme, the line-level redundant write have reduced the bit-flip caused by encryption to 24%. In accompany of a horizontal wear-leveling algorithm based on Start-Gap paper, they can achieve a

uniform line level write[22].

The paper Silent Shredder studies data shredding for operating system, an action to zero out physical pages before mapping them to a new process. It is a common practice in DRAM to secure the data but is highly undesirable for PCM where writes are expensive and life-costing. Therefore, for encrypted PCM, Silent Shredder repurposes the initialization vector to eliminate the writes for data shredding by resetting the associated counters before remapping and,by making the remaining data meaningless to processes that reuse them[24]. Not only does it reduce the writes by 48%, it also significantly speeds up read speed. This paper together with the DEUCE paper show different angles of how to take advantage of secure metadata to improve the lifespan of encrypted PCM[24].

Apart from improving the performance of encrypted PCM, significant efforts have been paid to improve slow writes and reduce power-consuming writes in regular PCM[26, 27, 28, 29, 30]. For the slow write problem, most studies fall in two categories. Either they focus on how to cancel, pause or truncate writes or they attempt to compress the write content[26, 27, 28, 33]. Power budget aware write scheduling and read-before-write comparison both want to reduce the expensive power consumption for write operation[29, 30]. All of these are very smart solutions. However, the problems they intend to conquer are not the foci of this dissertation.

Lastly, the difference between PCM and DRAM is summarized. In terms of structure, each DRAM cell is build on a capacitor and a transistor whereas PCM is based on phase change materials that show a huge resistance contrast between the amorphous phase and crystallized phases. PCM write speed depends on how fast its material can convert to and from amorphous to crystalline stages, where the high-resistance amorphous status is called RESET stage and the low-resistance crystalline stage is called SET stage. PCM read is performed when the material is under a small current pulse which would not change its status[34]. Experiments conducted by the research laboratories in industry show that, from the major metrics in performance evaluation, currently PCM is

still 4 to 100-fold slower than DRAM in terms of read speed[35]. The read bandwidth of PCM is also 4 to 10-fold smaller than that of DRAM. The minimum write latency and bandwidth of PCM is about 100-fold slower than its DRAM counterpart. However, the density of PCM is around four times higher than DRAM and its per-byte price is lower than DRAM[35]. DRAM needs to be repeatedly charged to maintain the status of its data while PCM does not. As such, PCM saves more power. Again, PCM shows a much higher risk of data leakage due to its non-volatility. Given the many difference between these two memory technologies, any well-suited designs and solutions for DRAM may not be relevant for PCM and may even bring interesting and practical problems.

Recovery for Secure Non-volatile Memory

In Chapter 2, we mainly discuss our approach to solve the data recovery problem for encrypted NVM. To be clear, when without notification, in the following chapters NVM only refers to PCM. As a feature of security concern, we pair the NVM with counter-mode encryption. Using counter-mode encryption has shown shortened latency in comparison to other methods, such as direct encryption[36]. To implement the counter-mode encryption, a specific area of memory is reserved for counters. Any data stored on secure NVM is encrypted with its own counter, or more accurately, a derived form of its counter called an initialization vector (IV). The memory controller is usually equipped with an AES or a type of cryptographic engine, which is used to generate an one-time pad (OTP) with an IV and a processor-specific key as the parameters. For plain text data, XORing with an OTP makes it encrypted and, therefore, could be stored on the PCM memory. On the other hand, to reverse back to the plain text, encrypted data from the memory has to be XORed again using the same OTP derived from same IV and the key. Note that only plain text is passed to last level cache for further processing in the processor. This scheme with its core components is shown in Figure 1.1.

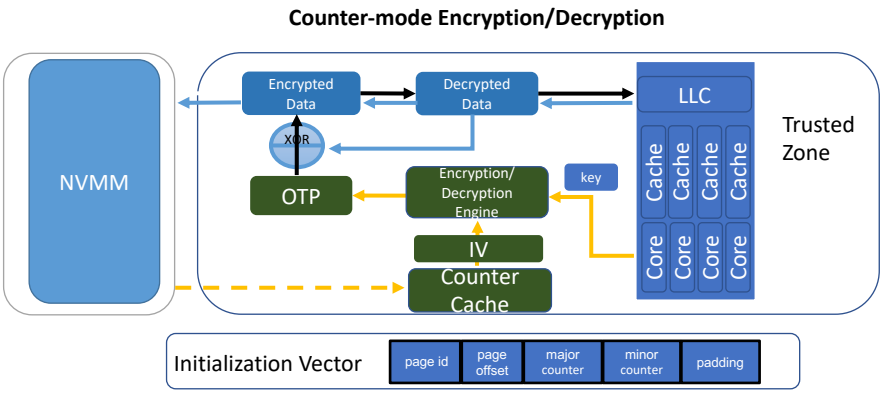

Figure 1.1: Counter-mode encryption in memory

Importantly, secure NVM relies on integrity trees, such as the Merkle Tree for data integrity verification[37]. Briefly, a Merkle Tree is a hashed tree with data and counters as its leaf nodes and with the hashed value of the leaves as its intermediate nodes all the way up to the root. The root is saved in a register that is kept safe in the processor, the trust-based zone for all of the hardware[37, 38]. Therefore, the intermediate hash nodes are also considered a part of the secure metadata that need to be reserved. Bonsai Merkle Tree significantly reduces the level and storage overhead of the original Merkle Tree by being built on hash values(MAC value) of only counters. Note that Merkle Tree is used to detect replay attack and counters actually work as a version of its associated data. MAC value itself can detect splicing and spoofing attacks. Therefore, it can be proved that data does not need Merkle Tree protection as long as it satisfies three conditions together: 1) A data chunk is protected by a MAC value. 2) A MAC value is calculated over the counter and the data address. 3) The counter values are under integrity protection[37]. A figure provides a more detailed illustration of how the Merkle Tree and Bonsai Merkle Tree are

constructed (Figure 1.2). In this dissertation, we acknowledge the importance of the presence of Merkle Tree and describe its role in data recovery.

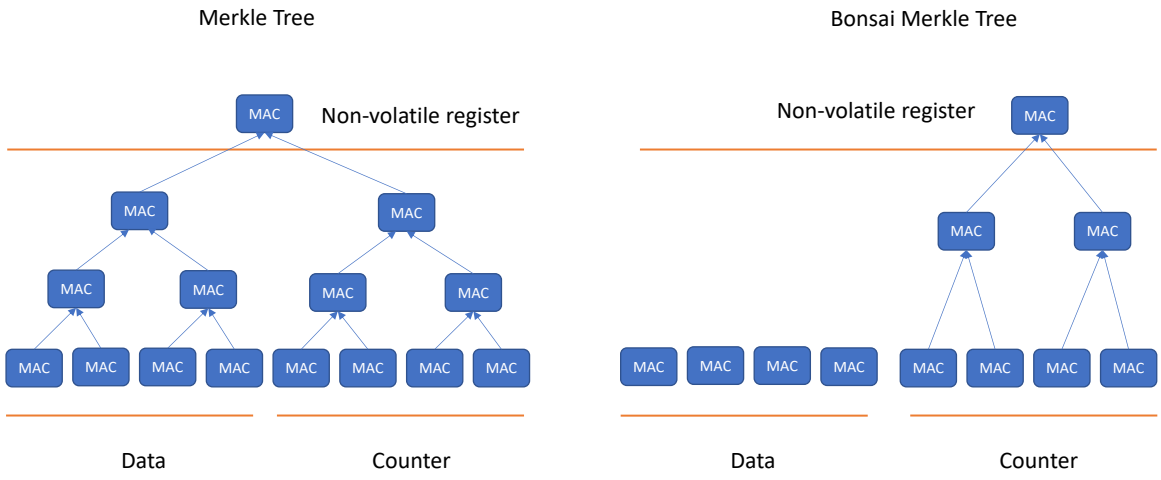

Figure 1.2: Demo construction of Merkle Tree and Bonsai Merkle Tree

In a previous research, it was discovered that in secure NVM after a crash, the counters and data that are persisted in the memory are not always matched with each other due to the incapability to flush all of the latest counter values to the memory[39]. Consequently, some of the data in the memory will forever lose their encryption counter values even if the power is switched on again. Data decryption failure makes data restoration meaningless. In that work, the authors provided a solution that uses an additional ready bit for both the counter and data to guarantee that they can track each other in an atomic way in write queues. The write queues blocks all the inconsistent data and counter pairs (ready bit not set) from being persisted. When a crash happens, write queues only drain the counter and data that both have arrived in the queues and have ready bit set. Moreover, to prevent performance degradation, the authors provide a recovery scheme only selectively persists data determined by programmers[39]. Research from another work utilized a

write-through counter cache for the same problem and it proposed to reduce counter writes through cancelling earlier write requests in the write queue if a recent request shares the same physical address with the pending requests[40]. Considering the write-endurance limitation of PCM and with the concerns of unavailable battery support, in Chapter 2, we discuss how to re-purpose ECC bits at a low cost to enable full memory data recovery without exacerbating the writes.

In secure NVM, multiple counter schemes are employed. Although the state-of-art counter scheme is proposed as a split counter, local and global counters are still discussed and employed in many studies[37, 3, 41]. Different types of counter scheme bring new challenges for counter persistence and recovery. Therefore, in Chapter 3, for a counter type not discussed in Chapter 2, we devised an epoch-based scheme for its persistence which only incurs a small number of extra write[42]. For data recovery, this newly devised scheme still relies on re-purposed ECC bits for stale counter identification.

CHAPTER 2: OSIRIS

As discussed in the Chapter 1, the representative type of NVM for this dissertation, phase change memory, demonstrates promising characteristics, such as low latency, high densities, high scalability and byte addressibility. Hence, it is considered a strong candidate to replace DRAM. Recently Intel announced an NVM product using 3D-XPoint technology, further confirming the market's interests in this emerging memory device. However, it has several drawbacks, i.e. limited write endurance, power-consuming writes and the data remanence problem due to the non-volatility. These drawbacks render it vulnerable to a variety of attacks, including bus snooping attacks, cold-boot attacks, and others. Therefore, NVM is usually paired with encryption to defend against the threats listed above.

Despite the use of the encryption mode, the persistence of security metadata triggers a problem when a crash happens. It is always guaranteed that the data in the cache is flushed to the main memory. However, the metadata(encryption counters) in the cache is not always persisted to the main memory in time; hence, some up-to-date metadata is lost during the crash. This leads to the failure to recover their matched encrypted data stored in the memory because the metadata remaining in the memory for these data chunks are stale.

Previous research solves this problem by strict and relaxed persistence of the security metadata. Although strict persistence guarantees the data recovery, the expense is huge as it either requires a write-through scheme that introduces double writes, or through a battery-supported write-back scheme which is not always affordable. Since write endurance is already a problem for NVM and the encryption exacerbates it, double writes is not desirable for persisting NVM. Relaxed persistence, despite reducing the number of writes significantly, requires efforts from programmers and may invite plain-text replay attacks.

In this work, we propose a hardware solution that can effectively persist the security metadata to

the main memory with fewer writes than a write-back scheme and does not need external power supply.

Background and Motivation

In this section, we discuss the main concepts that are relevant to our proposed solution, followed by motivation for our work.

Background

In this part of the chapter, we will discuss emerging NVMs and state-of-the-art memory encryption implementations.

Emerging NVMs

Emerging NVM technologies, such as Phase-Change Memory (PCM) and Memristor, are promising candidates to be the main building blocks of future memory systems. Vendors are already commercializing these technologies due to their many benefits. NVMs' read latencies are comparable to DRAM while promising high densities and potential for scaling better than DRAM. Furthermore, they enable persistent applications. On the other hand, emerging NVMs have limited, slow and power consuming writes. NVMs also have limited write endurance. For example, PCM's write endurance is between 10-100 million writes[23]. Moreover, emerging NVMs suffer from a serious security vulnerability: they keep their content even when the system is powered off. Accordingly, NVM devices are often paired with memory encryption.

Memory Encryption and Data Integrity Verification

There are different encryption modes that can be used to encrypt the main memory. The first one is the *direct encryption* (a.k.a ECB mode), where an encryption algorithm, such as AES or DES, is used to encrypt each cache block when it is written back to memory and decrypt it when it enters the processor chip again. The main drawback of direct encryption is system performance degradation due to adding the encryption latency to the memory access latency (the encryption algorithm takes the memory data as its input). The second mode, which is commonly used in secure processors, is *counter mode* encryption. In counter-mode encryption, the encryption algorithm (AES or DES) uses an *initialization vector* (IV) as its input to generate a one-time pad (OTP) as depicted in Figure 2.1. Once the data arrives, a simple bitwise XOR with the pad is needed to complete the decryption. Thus, the decryption latency is overlapped with the memory access latency. Counter-mode encryption have different design schemes. In state-of-the-art designs [38],each IV consists of a unique ID of a page (to distinguish between swap space and main memory space), page offset (to guarantee different blocks in a page will get different IVs), a per-block *minor* counter (to make the same value encrypted differently when written again to the same address), and a per-page *major* counter (to guarantee uniqueness of IV when minor counters overflow).

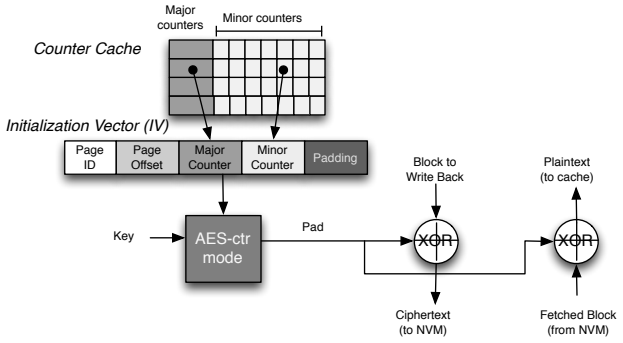

Figure 2.1: State-of-the-art counter mode encryption. AES is shown but other cryptographic algorithms are possible.

Similar to prior work [24, 39, 37, 38, 22], we assume counter mode processor-side encryption. In addition to hiding the encryption latency when used for memory encryption, it also provides strong security defenses against a wide range of attacks. Specifically, counter-mode encryption prevents snooping attacks, dictionary-based attacks, known-plaintext attacks and replay attacks. Typically, the encryption counters are organized as major counters (shared between cache blocks of the same page) and minor counters that are specific for each cache block [38]. This organization of counters can fit 64 cache blocks' counters in a 64B block; 7-bit minor counters and a 64-bit major counter. The major counter is only incremented when one of its relevant minor counters overflows, in which the minor counters will be reset and the whole page will be re-encrypted using the new major counter for building the IV of each cache block of the page[38]. When the major counter of a page overflows (64-bit counter), the key must be changed and the whole memory will be re-encrypted with the new key. This scheme provides a significant reduction of memory re-encryption rate and minimizes the storage overhead of encryption counters when compared to other schemes such as monolithic counter scheme or independent counter for each cache block. Additionally, a split-counter scheme allows for better exploitation of spatial locality of encryption counters, achieving a higher counter cache hit rate. Similar to state-of-the-art work [24, 43, 44, 38], we use a split-counter scheme to organize the encryption counters.

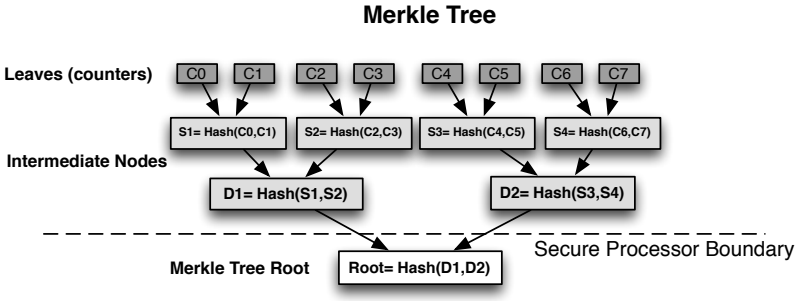

Figure 2.2: An example Merkle Tree for integrity verification.

Data integrity is typically verified through a Merkle Tree — a tree of hash values with the root maintained in a secure region. Actually, not only data but also encryption counter integrity needs to be maintained. As such, state-of-the-art designs combine both data integrity and encryption counters integrity through a single Merkle Tree (Bonsai Merkle Tree [37]). As shown in Figure 2.2, Bonsai Merkle tree is built around the encryption counters. Data blocks are protected by a MAC value that is calculated over the counter and the data itself. Note that only the root of the tree needs to be kept in the secure region, other parts of the Merkle Tree are cached on-chip to improve performance.

MAC are indispensable to detect spoofing and splicing attacks towards memory data[37]. MAC value is originally calculated for data and counters respectively in the context of secure memory. But later MAC is calculated only over data as long as its counter integrity is guaranteed via a Merkle Tree. This is because a MAC value is calculated over the input consisting of a counter and its cipher-text [37]. Despite sharing some similarity, the algorithms for commonly used hash and MAC are different. SHA-1 and SHA-256 are widely used hash functions and both take 80 rounds to process a 512-bit block[45]. For MAC computation, some schemes are based on hash function, such as HMAC, while others are not[46, 47]. HMAC literally means has-based message authentication code, which employs two passes of hash computation (e.g, any of the SHA algorithms) on the data size defined by the selected algorithm and its security strength solely relies on the length of secret key[46]. Used widely in network protocols, HMAC value can be truncated in practical[46]. GMAC is the NIST recommended authentication method that simply uses the Advanced Encryption Standard by default on 128-bit block and a secret key to encrypt the input data. The encryption itself is initialization vector based that is made of a 32-bit counter appending to a 96-bit initialization vector for randomness[47].

Encryption Metadata Crash Consistency

While crash consistency of encryption metadata has been overlooked in most memory encryption work, it becomes essential in persistent memory systems. If a crash happens, the system is expected to recover and restore its encrypted memory data. The steps needed to ensure crash consistency are depicted in Figure2.3.

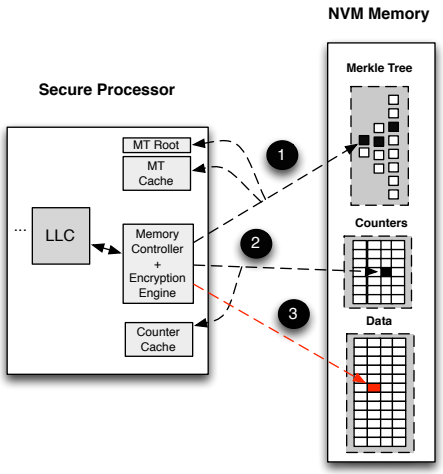

Figure 2.3: Steps for write operations to ensure crash consistency.

As shown in the Figure 2.3, when there is a write operation to NVM, first we need to update the root of the Merkle tree (as shown in step ①) and any cached intermediate nodes inside the processor. Note that only the *root* of the Merkle Tree needs to be kept in the secure region. In fact, maintaining intermediate nodes of the Merkle Tree can speed up the integrity verification. Persisting updates of intermediate nodes into memory is optional as it is feasible to reconstruct them from leaf nodes (counters) and then generate the root and verify it through comparison with that kept inside the processor. We stress that the root of the Merkle Tree must persist safely across system failures, e.g., through internal processor NVM registers. Persisting updates to intermediate nodes of the Merkle Tree after each access might speed up recovery time by reducing the time of

rebuilding the whole Merkle Tree after crash. However, the overheads of such a scheme and the infrequency of crashes make rebuilding the tree a more reasonable option.

In step ② (Figure 2.3), the updated counter block will be written back to memory as it gets updated in the counter cache. Unlike Merkle Tree intermediate nodes, counters are critical to keep and persist, otherwise the security of the counter-mode encryption is compromised. Moreover, losing the counter values can result in the inability to restore encrypted memory data. As noted by Liu et al. [39], it is possible to just persist counters of persistent data structures (or a subset of them) to enable consistent recovery. However, this is not sufficient from a security point of view; losing counters' values, even for non-persistent memory locations, can cause reuse of an encryption counter with the same key, which can compromise the security of the counter-mode encryption. Furthermore, legacy applications may rely on OS-level or periodic application-oblivious checkpointing, making it challenging to expose their persistent ranges to the memory controller. Accordingly, a secure scheme that persists counter updates and does not require software alteration is needed. Note that even for non-persistent applications, counters must be persisted on each update or the encryption key must be changed and all of the memory must be re-encrypted with a new key. Moreover, if the persistent region in memory is large, which is likely in future NVM-based systems, most memory writes will naturally be accompanied by an operation to persist the corresponding encryption counters, making step ② a common event.

Finally, the written block will be sent to the NVM as shown in step ③. Some portions of step ① and step ② are crucial for correct and secure restoration of secure NVMs. Also note that when updating the root of the Merkle Tree on the chip, updating the counter and writing the data are assumed to happen atomically, either using three internal NVM registers to save them before trying to update them persistently or using hold-up power that is sufficient to complete three writes. To avoid incurring high latency to update NVM registers for each memory write, a hybrid approach can be used where three volatile registers can be backed with hold-up power enough to write them

to the slower NVM registers inside processor. Ensuring such write atomicity is beyond the scope of this chapter; our focus is to avoid frequent persistence of updates to counter values in memory and using the fast volatile counter cache while ensuring safe and secure recover-ability.

Design

We first discuss our threat model, followed by the design of Osiris and the possible design options and trade-offs.

Threat Model

Our assumed threat model is similar to state-of-the-art work on secure processors [22, 24, 39, 44]. The trust base includes the processor and all its internal structures. Our threat model assumes that an attacker can snoop the memory bus, scan the memory content, try to tamper with the memory content and replay old packets. Differential power and electromagnetic inference attacks, as well as attacks that try to exploit processor bugs in speculative execution, such as Meltdown and Spectre, are beyond the scope of this chapter. Such attacks can be mitigated through more aggressive memory fencing around critical code to prevent speculative execution. Finally, our proposed solution does not preclude secure enclaves and hence can operate in untrusted Operating System (OS) environments.

Attack on Reusing Counter Values for Non-Persistent Data: While state-of-the-art [39] work relaxes persisting counters for non-persistent data, it introduces serious security vulnerabilities. Specifically, assume an attacker application uses known-plaintext and write it to memory, however, if the memory location is non-persistent the encrypted data will be written to memory but probably not the counter. Thus, by observing the memory bus, the attacker can find out the encryption pad by XORing the observed ciphertext, $(E_{key}(IV_{new}) \oplus Plaintext)$, with the $Plaintext$. Note that

it is also easy to predict the plaintext for some accesses, for instance, zeroing at first access. By now, the attacker knows the value of $E_{key}(IV_{new})$. After a crash, however, the memory controller will read IV_{old} and increment it, which generates IV_{new} and then encrypt the new application data written to that location to become $E_{key}(IV_{new}) \oplus Plaintext2$. Knowing $Plaintext2$ only needs XORing the ciphertext with the previously observed $E_{key}(IV_{new})$. Note that the stale counter could have been already incremented multiple times before the crash, hence multiple writes of the new application can reuse counters with known encryption pads. Note that such an attack only needs a malicious application to run (or just predictable initial plaintext of an application) and having a physical attacker or bus snooper.

Design Assumption

Counter and MAC are the major secure metadata in current secure memory system. They each serve different purpose. Counters are used for preventing data replay attacks, while MAC values can detect data splicing and spoofing attacks[38]. While there is no argument about where to place counters, MAC values can either be co-located with data on ECC chips, or they are located on the data chips just like counters are[38]. Although MAC is frequently discussed in research community, in commercial products, only ECC chips are normally paired with data chips on memory. Therefore, in this work, we assume MAC will be co-located with data with an extra burst. That requires the change of bus-width.

Design Options

Before delving deep into the details of Osiris, let's first discuss the challenges of securely recovering the encryption counters after a crash happens. Without a mechanism to persist encryption counters, once a crash occurs, you are only guaranteed that the root (kept in processor) of the

Merkle Tree is updated and reflects the most recent counter values written to memory; any write operation before being sent to NVM will update the affected parts/branches of the Merkle Tree up to the root. Note that most likely the affected branches of the Merkle Tree will be cached on the processor and there is no need to persist them as long as the processor can maintain the value of the root after crash. Later, once the power is back and we want to restore the system, we may have stale counter values in the NVM and stale intermediate values of the Merkle Tree.

Once the system is powered back, any access to a memory location needs two main steps: ① Obtaining the corresponding most-recent encryption counter from memory. ② Verifying the integrity of data through MAC and the integrity of the used counter value through Merkle Tree. As it is possible that Step ① results in a stale counter, Step ② will fail due to Merkle Tree mismatch. Remember that the root of the Merkle Tree has been updated before crash, thus using any stale counter will be detected. As soon as the error is detected, the recovery process stops. One naïve approach would be to try all possible counter values and use Merkle Tree to verify each value. Unfortunately, such a brute-force approach is impractical due to several reasons. First, finding out the actual value requires trying all possible values for a counter paired with calculating the corresponding hash values to verify integrity, which is impractical for typical encryption counter schemes where there could be 2^{64} possible values for each counter. Second, it is very unlikely that only one counter value is stale; many updated counters in the counter cache will be lost. Thus, reconstructing the Merkle Tree will be almost impractical if there are multiple stale counters. Let's say counters of blocks X and Y are lost, then we need to reconstruct Merkle Tree with all possible combinations/values of X and Y and then compare the resulting root with the one safely stored in processor. While for simplicity we only mention losing two counters, in actual crash where a counter cache is hundreds of kilobytes, we will likely have thousands of stale blocks.

Losing encryption counter values renders reconstructing Merkle Tree nearly impossible. Approaches such as brute-force trial of possibly lost counter values to reconstruct Merkle Tree will

likely take impractical time especially when multiple counter values have been lost. Hence, verifying the integrity/correctness of the counters stored in NVM is challenging.

One possible way to reduce reconstruction time is through employing ***stop-loss*** mechanisms to limit the number of possible counter values to verify for each counter after recovery. Unfortunately, since there is no way to pinpoint exactly which counters have lost their values, an aggressive searching mechanism is still needed. If we limit the number of writes to each counter block before persisting it to only N, then we need to try up to N^S combinations for reconstruction where S is the number of data blocks. For instance, let's assume we have a 128GB NVM memory and 64B cache blocks, then we have 2G blocks. If we only set N to 2, then we need up to $2^{2^{31}} = 2^{2147483648}$ trials. Accordingly, stop-loss mechanism could reduce the time to reconstruct the Merkle Tree, however, still is impractical.

Obviously, a more explicit confirmation is needed before proceeding with an arbitrary counter value to reconstruct the Merkle Tree. In other words, we need a hint on what was the most recent counter value for each counter block. For instance, if the previously discussed stop-loss mechanism is used along with an approach to bookkeep the *phase* within the N trials before writing the whole counter block, then we can start with a more educated guess. Specifically, each time we update a counter block N times in the counter-cache, we need to persist its Nth update in the memory, which means that we need $log_2 N$ bits (i.e., phase) for each counter block be written atomically with the data block. Later, the system starts recovery, it knows the exact difference between the most recent counter value and one used to encrypt the data through the phase value. Co-locating the data blocks with a few bits that reflect most-recent counter value used for encryption can enable fast-recovery of the counter-value used for encryption. Note that if an attacker tries to replay old data along with their phase bits, then the Merkle Tree verification will detect the tampering due to mismatch in the resulting root of the Merkle Tree. Although stop-loss along with phase storage can make the recovery time practical, adding more bits in memory for each cache line is tricky for

several reasons. First, as discussed in [39], increasing the bus-width requires adding more pins to the processor. Even avoiding extra pins by adding extra burst in addition to the 8 bursts of 64-bit bus width for each 64B block is expensive and requires support from DIMMs in addition to under utilization of data bus (only few bits are written in the 64-bit wide memory bus in the last burst). Second, major memory organization changes are needed, e.g., row-buffer size, memory controller timing and DIMM support. Additionally, cache blocks are no longer 64B aligned, which can cause complexity in addressing. Finally, extra bit writes are needed for each cache line to reflect the counter phase, which can map to a different memory bank, hence additional occupation of bank for write latency.

To retain the advantages of stop-loss paired with phase bookkeeping but without extra bits, Osiris re-purposes already existing ECC bits as a fast counter recovery mechanism. The following sub-section will discuss in details how Osiris can elegantly employ ECC bits of data to find out the counter used for encryption.

Design

Osiris mainly relies on inferring the correctness of an encryption counter from calculating the ECC associated with the decrypted text and compares it with that encrypted and stored along with the encrypted cache line. In conventional (not encrypted) systems, when the memory controller writes a cache block to the memory, it also calculates its ECC, e.g., hamming code, and stores it along with the cache line. In other words, the tuple will be written to the memory when writing cache block X to memory is $\{X, ECC(X)\}$. In contrast, in encrypted memory systems, there are two options to calculate ECC: ① Using the plaintext, then encrypt it with the cache line before writing both to memory. ② The second option is to encrypt the plaintext, then calculate the ECC over the encrypted block before both are written to the memory. Although approach ② allows over-lapping decryption and ECC verification, most ECC implementations used in memory controllers,

e.g., SEC-DED Hsiao Code [48], take less than a nanosecond to complete [49, 50, 51], which is negligible compared to cache or memory access latencies [52]. Additionally, pipelining the arrival of bursts with decryption and ECC bits decoding will completely hide the latency. However, we observe that calculating ECC bits over the plaintext and encrypting it along with the cacheline can provide low-cost and fast way of verifying the correctness of the encryption/decryption operation. Specifically, in the counter-mode encryption, the data is decrypted using the following: $\{X, Z\} = E_{key}(IV_X) \oplus Y$, where Y is potentially the encryption of X along with its ECC and Z is potentially equal to $ECC(X)$. In conventional systems, if $ECC(X) \neq Z$, then the reason is definitely an error (or tampering) occurred on X or Z. However, when the counter-mode encryption is used, the reason could be an error (or tampering) occurred on X or Z, or *wrong IV* is used to do the encryption, i.e., decryption is not successful. When the ECC function is applied over the plaintext and the resulting ECC bits are encrypted along with the data, ECC bits can provide a sanity-check for the used encryption counter. Any tampering with the counter value will be detected by a *clear* mismatch of the ECC bits result from that invalid (wrong/stale counter) decryption; results of $E_{key}(IV_{old})$ and $E_{key}(IV_{new})$ are very different and independent. Note that in Bonsai Merkle Tree, data-integrity is protected through MAC values that are calculated over each data and its corresponding counter. While relying on ECC for sanity-checking the used counter can be used, the ECC bits can fail to provide guarantees as strong as cryptographic MAC values. Accordingly, we adopt Bonsai Merkle to additionally protect data integrity. However, ECC bits when combined with counter-verification mechanisms, can provide tamper-resistance as strong as the error detection of the used ECC algorithm. ***Important Note*:** Stream ciphers, e.g., CTR and OFP modes, do not propagate errors, i.e., an error in the ith encrypted data bit will result in an error in ith bit of decrypted data, hence the reliability is not affected. In encryption modes where an error in encrypted data results in completely unrelated decrypted text, e.g., block cipher modes, careful consideration is required as encrypting ECC bits can render them useless when there is an error. For our scheme, we focus on state-of-the-art memory encryption, which uses CTR-mode for

security and performance reasons.

The question is how to proceed when there is an error detected due to a mismatch between the expected and stored ECC. As the reader would expect, the first step is to find if the error is correctable using the ECC code. If the error is un-correctable, before giving up, we take the odds that the IV used is incorrect, i.e., the decryption was not successful. Our goal is to find out if the error is due to using a wrong IV. Below is a summary of the common reasons for such a mismatch between the expected ECC and the stored ECC after decryption:

Table 2.1: Common sources of ECC mismatch

Error Type	Typical Fix
Error on stored data	can be fixed if the error is correctable, e.g., single bit failure
Error on ECC	typically unrecoverable
Stale/Wrong IV	**Speculate the correct IV and verify it**

As shown in Table 2.1, one reason for such a mismatch is using an old IV value. To better understand how this can happen, we recall the counter cache persistence issue. If a cache block is updated in memory, it is necessary to also update and persist its encryption counter, for both security and correctness reasons. Given the ability to detect the use of stale counter/IV, we can *implicitly* reason about the likelihood of losing the most-recent counter value due to a sudden power loss. To that extent, in theory, we can try to decrypt the data with all possible IVs and stop when an IV successfully decrypts the block, i.e., the resulting ECC matches the expected one ($ECC(X) = Z$). At that point, there is a high chance that such an IV was actually the one used to encrypt the block, but was either lost due to inability to persist the new counter value after persisting the data, or due to a *relaxed scheme*. Osiris builds upon the later possibility using a relaxed counter persistence scheme employing ECC bits to verify the correctness of the counter used for encryption. As discussed earlier, it is impractical to try all the possible IVs to infer the one used to encrypt the block. Thus, Osiris deploys the stop-loss mechanism to limit such possibility to only N counter updates;

i.e., the correct IV should be within $[IV + 1, IV + N]$, where IV is the most recent IV that was stored/persisted in memory. Note that once the speculated/chosen IV passes the first check through ECC sanity-check it also needs to be verified through Merkle Tree.

We propose two flavors for Osiris, baseline ***Osiris*** and ***Osiris-Plus***. In the baseline, all counters being read from memory reflect their most-recent values during normal operation, and the most-recent value is either updated in cache or evicted/written-back to memory. Thus, inconsistency between counters in memory and counters in cache can happen due to crash or tampering with counters in memory. In contrast, Osiris-Plus strives to even outperform Battery-Backed Write-Back counter-cache scheme through purposely skipping counter updates and recovering their most-recent values when reading them back, hence inconsistency can happen during normal operation. Below is a further discussion on both baseline Osiris and Osiris-Plus.

Normal Operation Stage: During normal operation, Osiris adopts write-back mechanism by updating memory counters when evicted from the counter cache. Thus, Osiris can always find the most-recent counter value either in cache or by fetching it from memory in case of miss. Accordingly, Osiris operation in normal operation is similar to conventional memory encryption except that a counter is additionally persisted once each Nth update, hence acting like a write-through for the Nth update of each counter. In contrast, Osiris-Plus allows occasional dropping of most-recent values of encryption counters through relying on run-time recovery mechanism of the most-recent values of counters. In simple words, Osiris-Plus relies on trying multiple possible values on each counter miss to recover the most-recent one before verifying it through Merkle Tree, whereas baseline Osiris would do that only at system recovery time.

System Recovery Stage: During system recovery time, both Osiris and Osiris-Plus need to reconstruct Merkle Tree at the time of restoration. Furthermore, both need to use most-recent values of counters to reconstruct a Merkle Tree that eventually has a root that matches the root stored in the

secure processor. In both Osiris and Osiris-Plus, the system recovery will start with traversing all memory locations (cache lines) in integrity verified region (all memory in secure NVM). For each cache line location, i.e., 64B address, the memory controller uses ECC value after decryption as a sanity check of the counter retrieved from memory, however, if the counter value fails the check, all possible N values will be checked to find the most-recently used counter value. Later, the correct value will overwrite the current (stale value) counter in memory. After all memory locations are vetted, Merkle Tree will be reconstructed with the recovered counter values and eventually build up all intermediate nodes and the root. In the final step, the resulting root will be compared with that saved and kept in the processor. If a mismatch occurs, the data integrity of the memory cannot be verified and it is very likely that an attacker has replayed both counter and corresponding encrypted data+ECC blocks.

In the next parts, we will discuss the design of Osiris and Osiris+Plus by guiding the reader through the steps of read/write operations in both schemes.

Osiris Read and Write Operations

During a write operation, once there is a cache block evicted from LLC, the memory controller will calculate the ECC of the data in the block, as shown in Step ① (Figure 2.4). Note that write operations happen in the background and typically are buffered for a while in the write-pending queue. Later, in Step ②, the memory controller obtains the corresponding counter in case of miss and evict/write-back the evicted counter block, if dirty. The counter value obtained from Step ② will be used to proactively generate the encryption pad, as shown in Step ③. Later, in Step ④, the obtained counter value will be verified (in case of miss) and then the counter and affected Merkle Tree (including root) are updated in Merkle Tree cache. Additionally, unlike typical write-back counter-cache, if the new counter value is a multiple of N or 0, then the new counter value is also persisted before proceeding. Finally, in Step ⑤, the data+ECC is encrypted using the encryption pad and written to memory.

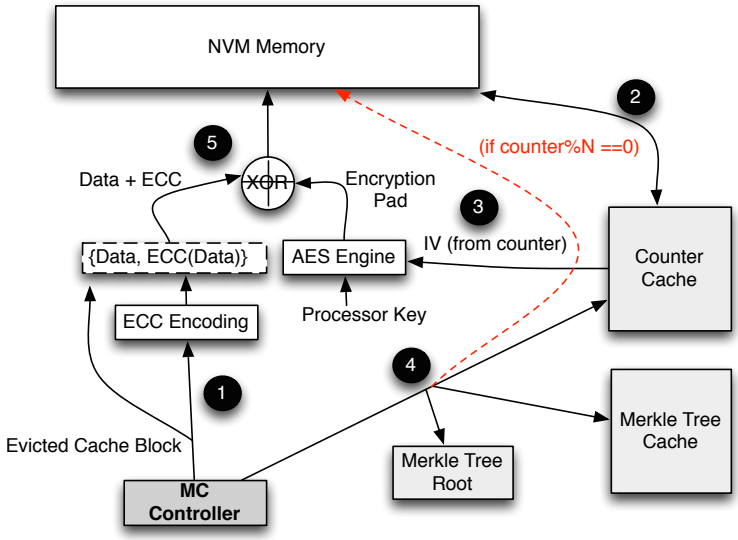

Figure 2.4: Osiris write operation

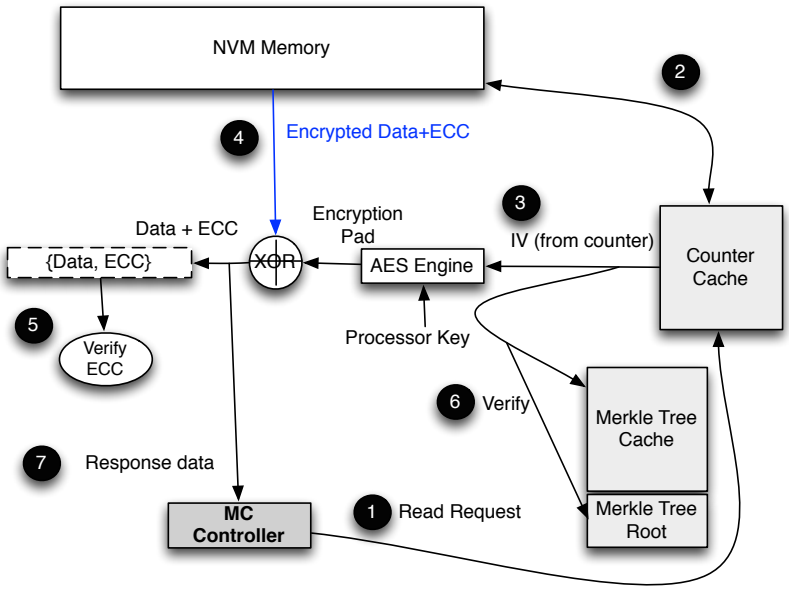

Figure 2.5: Osiris read operation

During read operation, the memory controller will obtain the corresponding counter value from counter cache (with hit) or memory (with miss) and evict the victim block, if dirty, as shown in Steps ① and ② (Figure 2.5). Later, the obtained counter value will be used to generate the encryption pad as shown in Step ③. In Step ④, the actual data block is read from memory and decrypted using the pad generated in Step ③. Later, in Step ⑤, traditional ECC checking occurs to the decrypted data. Finally, before proceeding, if the counter block is fetched from memory (miss), the integrity of the counter value is verified, as shown in Step ⑥. Finally, as shown in Step ⑦, the memory controller receives the decrypted data which is then forwarded to the cache hierarchy by the memory controller. Note that many of the steps can be overlapped with any conflicts, for instance, Step ⑥ and steps ④ and ⑤.

Osiris-Plus Read and Write Operations

The write operation in Osiris-Plus is very similar to the write-operation in baseline Osiris except that it does not write back evicted dirty blocks from counter cache (as could happen in Step ② of Figure 2.4); Osiris-Plus recovers the most-recent value of counter each-time it is read from memory and only updates it in memory each Nth update. The read operation of Osiris-Plus is demonstrated in Figure 2.6.

The main difference between Osiris and Osiris-Plus are Steps ⑤ and ⑥ in Figure 2.6. Osiris-Plus utilizes additional encryption engines and ECC units to allow fast recovery of the most-recent counter value. Note that given the fact that most AES encryption engines are pipelined and the candidate counter values are sequential, using a single encryption engine instead of N engines can only add N extra cycles. Later, once a counter value is detected, through post-decryption ECC verification, to be the most-recent one, it will be verified through Merkle Tree similar to baseline Osiris. Later, once counter is verified, the data resulting from decrypting the ciphertext with the most-recent counter value will be returned to the memory controller. Note that the recovered

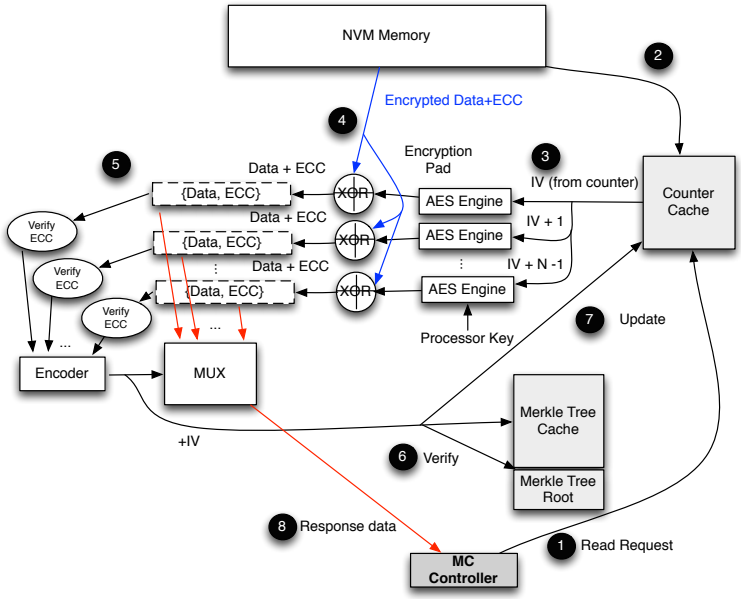

Figure 2.6: Osiris-Plus read operation

counter value is updated in the counter-cache with the recovered value, as shown in Step ⑦.

Reliability and Recovery from Crash

As mentioned earlier, to recover from crash, Osiris and Osiris-Plus need to first recover the most-recent counter values through utilizing post-decryption ECC bits to find the correct counters. Later, the Merkle Tree will be constructed and the root will be verified and compared with that kept in the root. While the process seems reasonably simple without errors, the recovery process can get complicated in the presence of errors. Specifically, uncorrectable errors in the data or encryption counters render generating a matching Merkle Tree root nearly impossible, and hence the inability to verify the integrity of memory. Note that such uncorrectable errors will have the same effect on integrity-verified memories even without deploying Osiris.

Uncorrectable errors in encryption counters protected by Merkle Tree can potentially fail the whole

recovery process. Specifically, when the system attempts to reconstruct the Merkle Tree, it will fail to generate a matching root. Furthermore, it is also infeasible to know which part of the Merkle Tree causes the problem; only the root is maintained and all other parts of Merkle Tree should work perfectly and generate the same root or none is trusted. One way to mitigate such single-point of failure is to keep other parts of the Merkle Tree in the processor and guarantee they are never lost from the secure processor chip. For instance, for an 8-ary Merkle Tree, the immediate 8 children of the root are also saved all the time in the processor. In a more capable system, the root, its immediate 8 children and their immediate children are kept, which is a total of 73 MAC values. In case recovery fails to produce the root of the Merkle Tree, we can look at which children are mismatching, and then declare that part of the tree as un-verifiable and probably warn the user/OS. While we provide such insights to solve this problem, we assume the system architects choose to only save the root, however, having more NVM registers inside the processors to save more levels of Merkle Tree can be implemented in case of high error systems. In fact, Triad-NVM addressed the importance and the performance overhead for persisting Merkle Tree. Persisting the lower levels of the Merkle Tree can significantly shorten the Merkle Tree recovery after the crash and, speed up the recovery process by 3 orders of magnitude[53]. Anubis, on the other hand, optimizes the Osiris by recording the addresses of updated counters in a reserved table[54]. Moreover, it further discusses how to recover a SGX-style integrity tree[54]. Both papers point out that when integrity tree is used for memory integrity purpose, rebuilding the tree is the most critical task and the time-bound job to guarantee the later recovery of secure metadata and actual data.

To formally describe our recovery process success rate, we can look at two cases. In the first case, when no errors occur in the data, since each 64B memory block has 8B ECC bits, the probability that a wrong counter results in correct (indicate no errors) ECC bits for the whole block is only $\frac{1}{2^{64}}$, i.e., similar probability to guessing a 64-bit key correctly, which is next to impossible. Note that each 64B block is practically divided into 8 64-bit words [55], each has its own 8-bit ECC, i.e.,

each bus burst will have 64-bit data and 8-bit ECC (total of 72 bits). This is why most ECC-enabled memory systems will have 9 chips per rank instead of 8, where each chip provides 8 bits. The 9th chip provides the ECC 8-bits for the burst/word. Also note that the 64B block will be read/written as 8 bursts on a 72-bit memory bus.

Bearing in mind the organization of ECC codewords for each 64B memory block, let's now discuss the case where there is actually an error in data.

For an incorrect counter, the probability that an 8-bit ECC codeword indicates that there is no error is $P_{no-error} = \frac{1}{2^8}$ and the probability of not indicating that there is no-error, i.e., that there is an error, is $P_{error} = 1 - \frac{1}{2^8}$. The probability that k codewords of the 8 ECC codewords indicate an error can be represented by a Bernoulli Trial as $P_k = \binom{8}{k} \times (1 - \frac{1}{2^8})^K \times (\frac{1}{2^8})^{8-K}$. For example, P_2 represents the probability of having 2 of the 8 ECC codewords flagging an error for a wrongly decrypted data and ECC (semi-randomly generated). Accordingly, if we use our metric to filter out encryption counters that have 4 or more ECC codewords indicating an error, then the probability of our success in detecting wrong counters can be given by $P_{k\geq 4} = 1 - (P_0 + P_1 + P_2 + P_3)$. We find that $P_{k\geq 4}$ is nearly 100%. Even $P_{k\geq 7}$ is 99.95%. Note that the probability all 8 codewords indicate an error is 96.91%, which is still very high. In other words, Osiris can successfully identify wrong counters with a success rate of almost 100% by filtering out any counter has 7 or more ECC codewords indicating errors. Thus, except for the pathological case where a cache block has actual errors on each of its words, Osiris can reliably distinguish wrong counters with a success rate of almost 100%. In that extremely pathological case, where each word of the memory block has at least one error, all counter values will be filtered out, and then Osiris can verify all the counters values (e.g., 8 values) through Merkle Tree to recover the correct one. Note that the probability a bit error occurs is very small, however, the described pathological case occurrence requires at least an error to occur on each of the 8 memory words in the block at the same time, which is extremely rare. Note that the words and ECC are typically interleaved and spread in the row and thus nearly

independent[55]. It is also important to note that all of our discussion here about detecting a wrong encryption counter in the presence of errors is relevant to the case of having an error.

In summary, Osiris can detect wrong counter reliably in all cases except the pathological case where there is at least an error *on each word* of the block corresponds to the counter being recovered, which will require an additional step of verification through Merkle Tree. Thus, Osiris does not affect how many errors an ECC can detect/correct per word but limits the number of faulty words within a 64B to not exceed 7 words to avoid additional step (Merkle Tree) before detection/correction. We believe that having errors on each word of a block is an extreme case and will not affect the adoption of Osiris. Note that some PCM prototypes use similar ECC organization but with larger number of ECC bits per 64-bit words, e.g., 16-bit ECC for each 64-bit word[56], which even makes our detection success rate even closer to perfect.

Different ECC Schemes

The ECC scheme that we employ is the SEC-DEC algorithm[48], one of the widely implemented encoding schemes for memory reliability that corrects 1-bit error. DEC-TEC code that handles multiple-bit(2-bit) error correction is another commonly seen scheme[45]. DEC-TEC codes requires 15 bits for each 64-byte data block. To incorporate DEC-TEC with Osiris, the memory bus width needs to increase another eight bits.

A tiered ECC system is an active direction of research. LOT-ECC, Virtualized ECC , Multi-ECC, Clean-ECC are the representative works in this area that not only provide reliability but concern memory access granularity[57, 58, 59, 60]. All these schemes use additional storage to provide redundant error protection. The similarity among these schemes is that they divide their ECC codes into different tiers that are respectively based on their functions, namely, error detection, error localization and error correction. Multi-ECC uses Reed-Solomon codes(16 bits) for error

detection and erasure correction, using complement checksum for error localization[59]. LOT-ECC was one of the earliest tiered ECC protection schemes. To eliminate a dedicated ECC chip, the error detection code is co-localized with data. Error reconstructions codes comprise the other three tiers. Virtualized ECC, in particular, after error detection, mitigates hardware errors in a software-visible memory pool[58]. Clean-ECC needs a regular 64-bit dbus plus eight bits redundancy. Apart from data chips, two redundant chips serve to store detection codes and correction codes, respectively[60]. Virtualized ECC and Clean-ECC are compatible with Osiris schemes, but due to different per-data vs ECC code length, the minor counter length might need to change. Multi-ECC may be compatible with Osiris, but more knowledge is required for checksum algorithm. Moreover due to the error correction codes, the use of encrypted or non-encrypted error detection codes is to be closely considered.

Other commonly used reliability schemes include but are not limited to parity and Chipkill correction codes[61, 45]. Parity is simple for error detection but lacks the capacity to correct errors. On the other hand, Chipkill-correction code can significantly reduce the uncorrectable error rate by up to more than tenfold (36x or 42x) in comparison to the SECDEC scheme from DRAM[62, 63, 64]

The Chipkill-correction code can guarantee that errors caused by a chip failure are corrected by ECC codes, and it can detect the errors on two different chips, but at the expense of activating four ranks with 18 chips per bank[65].

The assumption that Chipkill is based on is that multiple-bit error(up to four adjacent bits) has a large probability of taking place on one chip[65]. To be capable of fixing four adjacent bits, one implementation of Chipkill is that every four bits will be mapped to four different ECC words. To achieve this, the simplest solution is to interleave four SEC-DEC algorithms[65, 66]. However, this is incompatible with Osiris. The presumption in Osiris is that, every 64-byte data block with its encoded ECC is encrypted with the same counter. However, when 4 SEC-DEC algorithms

are interleaved, per 64-byte data is encoded by 4 different ECCs and therefore is difficult to be encrypted by a single counter. However, we do not rule out other implementations of Chipkill are not compatiable with Osiris. In commercialized chipkill-level products, not only can two failed devices be detected, but also one (SCCDCD) or two(Double Chip Sparing) failed devices can be corrected[67].

When NVM is used as storage, the most commonly used ECC schemes are BCH and LDPC[68]. However, counter-mode encryption is not used in SSD or flash disks and, therefore, is not discussed here.

Systems without ECC

Some systems do not employ ECC bits, but rather rely on MAC values that can be co-located with data instead of ECC bits [44]. For instance, the ECC chips in the DIMM can be utilized to store the MAC values of the Bonsai Merkle Tree to allow obtaining data integrity verification MACs along with the data. While our description of Osiris and Osiris-Plus was focused on using ECC, MAC values can also be used to achieve the exact same purpose; a sanity-check for the decryption counter used. The only difference is that if there is an error, when ECC bits are used, we can use the number of mismatched ECC bits as a way to guess the counter value, whereas MAC values tend to differ significantly when there is any error in data. Accordingly, to mitigate this problem, MAC values that are aggregations of multiple MACs per word, e.g., 64 bits that are made of eight bits for each 64 data bits, can be used; in this way, the difference between the generated MACs and the stored MACs for different counter values can be used as a selection criteria for the candidate counter value. Note that our proposed schemes also work with ECC and MAC co-designs such as Synergy(parity+MAC)[44].

System with both ECC and MAC

As discussed above, some recent research work studies the co-existence of ECCs and MACs[44, 45]. As a matter of fact, none of these works take advantage of ECCs; instead, they either use MAC or parity codes. When a MAC value is required for other types of data security guarantees, accessing MACs and ECCs at the both memory access is difficult at the current stage without any change. However, increasing the bus width to 80b or 144b can easily solve the extra memory access problem. Especially for NVM, which, due to high error rates, requires three ECC chips for each 8-data chip, increasing the bus width is obvious as multiple memory access is required even without MAC values.

In any event, the Osiris scheme is still effective, but at the expense for one more read/write for a data MAC value. This means the write benefit might be reduced for Osiris scheme. However, to compensate for this, we can increase Osiris phase change number to reduce the frequency of persistence. Nevertheless the performance benefit may not reduce significantly as currently the memory controller has multiple channels, simultaneously allowing the parallel memory accesses at the same time. It also depends that on how the MAC and data is co-located and how long the MAC value is used for each piece of data. According to [37], for every data cacheline fetching, four MACs are fetched on another cacheline. Therefore, caching is helpful to reduce memory writes under such circumstances.

Security of Osiris and Osiris-Plus

Since Osiris and Osiris-Plus rely on final verification step through Merkle Tree, Osiris and Osiris-Plus have security guarantees and tamper-resistance similar to any scheme that uses Bonsai Merkle-Tree, such as state-of-the-art secure memory encryption schemes [22, 44, 24, 39].

Evaluation

In this section, we evaluate the performance of Osiris with other state-of-the-art persisting cache designs. Section 2 describes the methodology we use for the experiments. Section 2 discusses the performance impact of the design for Osiris/Osiris Plus in terms of limit value and the performance comparison with other cache designs.

Methodology

Table 2.2: Configuration of the simulated system.

Processor	
CPU	4-core, 1GHz, out-of-order x86-64
L1 Cache	private, 2 cycles, 32KB, 2-way, 64B block
L2 Cache	private, 20 cycles, 512KB, 8-way, 64B block
L3 Cache	shared, 32 cycles, 8MB, 64-way, 64B block
DDR-based PCM Main Memory	
Capacity	16 GB
PCM Latencies	60ns read, 150ns write [69]
Organization	2 ranks/channel, 8 banks/rank, 1KB row buffer, Open Adaptive page policy, RoRaBaChCo address mapping
DDR Timing	tRCD 55ns, tXAW 50ns, tBURST 5ns, tWR 150ns, tRFC 5ns [69, 39] tCL 12.5ns, 64-bit bus width, 1200 MHz Clock
Encryption Parameters	
Counter Cache	256KB, 16-way, 64B block

We model Osiris in Gem5 [70] with the system configuration presented in the Table 2.2. We implement a 256KB, 16-way set associative counter cache, with a total number of 4K counters. To stress-test our design, we select memory-intensive applications. Specifically, we select eight memory-intensive representative benchmarks from the SPEC2006 suite [71] and from proxy applications provided by the U.S. Department of Energy (DoE) [72]. The goal is to evaluate the performance and the write traffic overheads of our design. In all experiments, the applications are fast-forwarded to skip the initialization phase, and then followed by the simulation of 500 million instructions. Similar to prior work [43], we assume the overall AES encryption latency to be 24 cycles, and we overlap fetching data with encryption pad generation. Below are the schemes we

use in our evaluation:

No-encryption Scheme: The baseline NVM system without memory encryption.

Osiris Scheme: Our base solution that eliminates the need for battery while minimizing the performance and write traffic overheads.

Osiris Plus Scheme: An optimized version of the Osiris scheme that additionally eliminates the need for evicting dirty counter blocks, however, at the cost of extra online checking mechanism to recover the most recent counter value.

Write-through (WT) Scheme: A strict counter-atomicity scheme that, for each write operation, enforces both counter and data blocks to be written to NVM memory. Note that this scheme does not require any unrealistic battery or power supply hold-up time.

Write-back (WB) Scheme: A battery-backed counter cache scheme. The WB scheme only writes to memory dirty evictions from counter cache. However, WB scheme assumes a battery is sufficient to flush all dirty blocks in counter cache.

Analysis

As discussed before, the purpose of Osiris/Osiris-Plus is to persist the encryption counters in NVM memory in response to system crash recovery, however, with reduced performance overhead and write traffic. Therefore, the selection of number N for update interval (also discussed in Section 2) is critical in determining the performance improvement. As such, in this section, we study the impact of choosing different N (limit) for Osiris/Osiris-Plus on performance. Next, we present the performance analysis of multiple benchmarks in response to different persistent schemes discussed and compared in this chapter. Our evaluation results are consistent with the goal of our design for Osiris and Osiris-Plus.

Impact of Osiris Limit

To understand the impact of Osiris limit (also called N) on performance and write-traffic, we vary the limit in multiples of two and observe the corresponding performance and write-traffic. Figure 2.7 shows the performance of Osiris and Osiris-Plus normalized to no-encryption while varying the limit. The figure also shows WB and WT performance normalized to no-encryption to facilitate the comparison.

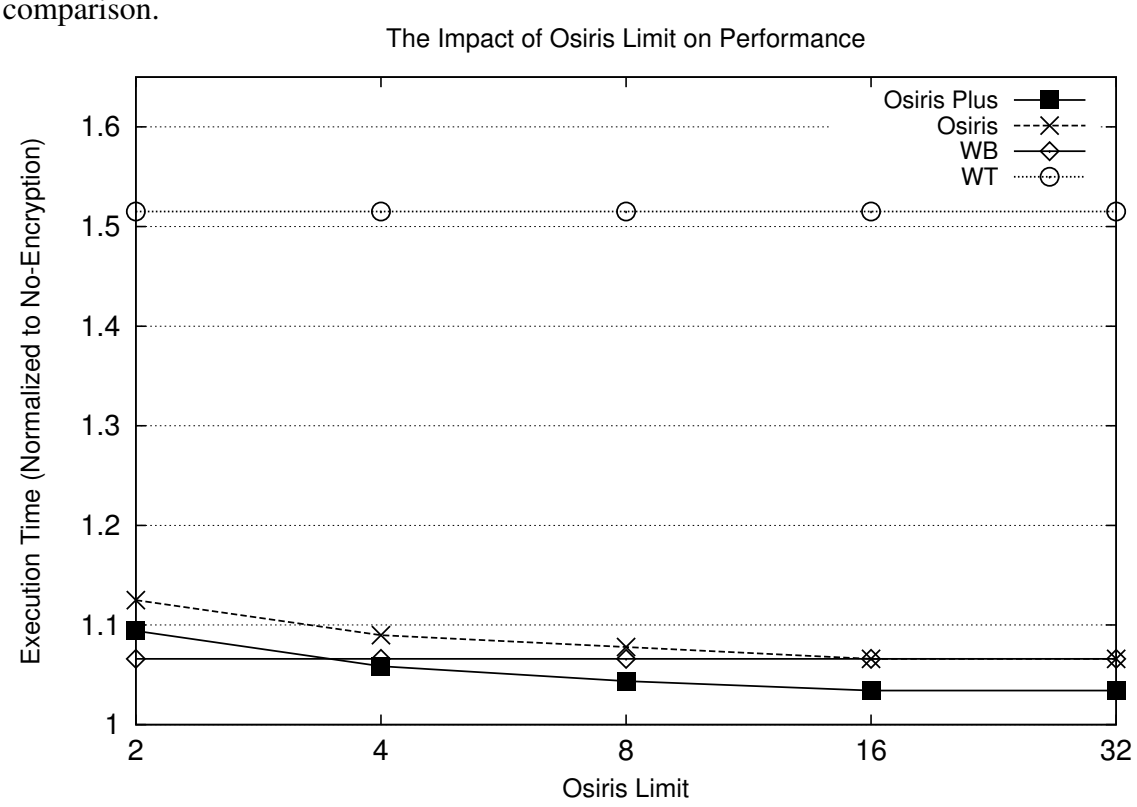

Figure 2.7: The impact of Osiris limit on performance.

From Figure 2.7, we can observe that both Osiris and Osiris-Plus benefit clearly from increasing the limit. However, as discussed earlier, having large limit values can cause increase in recovery time and potentially large number of encryption engines and ECC units (in case of Osiris-Plus). Accordingly, it is important to find the point which can no longer bring in justifiable gains if in-

creased. From the Figure2.7, we can observe that Osiris at limit 4 has an average performance overhead of 8.9% compared to 6.6% and 51.5% for WB and WT, respectively. In contrast, also at limit 4, Osiris-Plus has only 5.8% performance overhead, which is even better than WB scheme. Accordingly, we can observe that at limit 4, both Osiris and Osiris-Plus perform close to or outperform the WB scheme even though they do not need any battery or hold-up power. In large limit values, e.g., 32, Osiris performs similar to write-back; dirty blocks will be evicted before absorbing the limit of number of writes, hence the counter block is rarely persisted before eviction. In contrast, Osiris-Plus brings down the performance overheads to only 3.4% (compared to 6.6% for WB), but again at the cost of large number of encryption engines and ECC units.

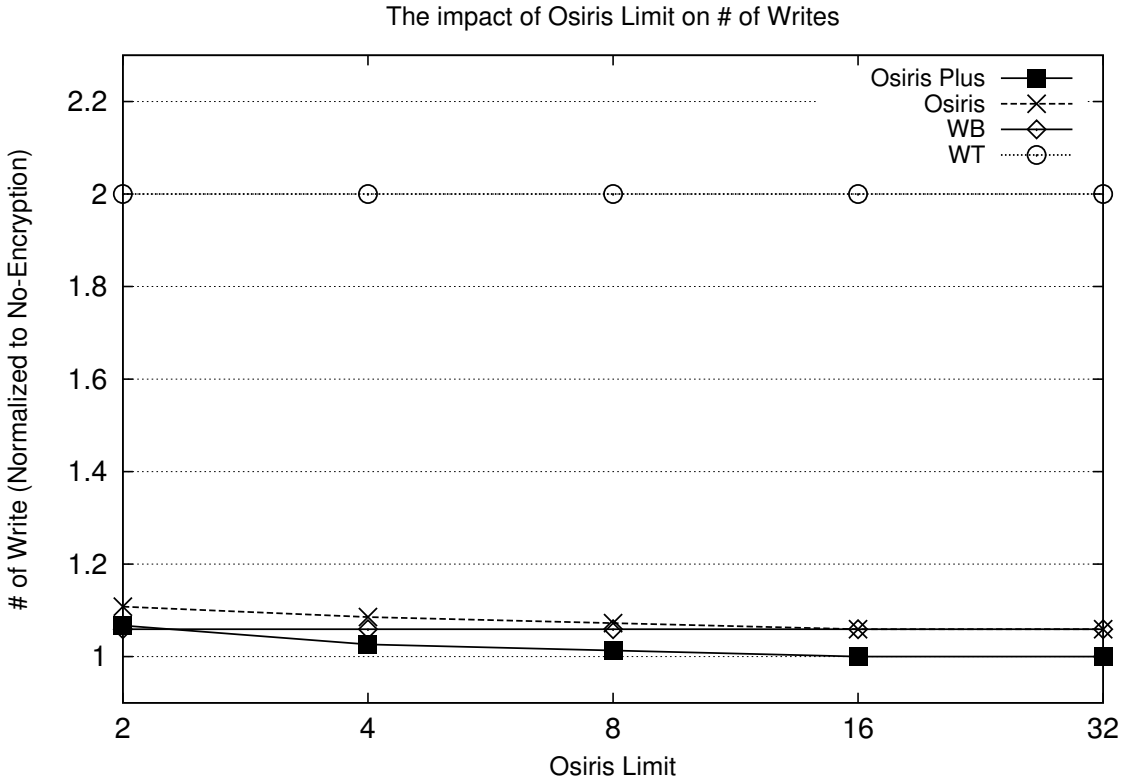

Figure 2.8: The impact of Osiris limit on number of writes.

A similar pattern can be observed in Figure 2.8 demonstrating less write traffic for both schemes.

At limit 4, Osiris has a write-traffic overhead of 8.5% whereas WB and WT have 5.9% and 100%, respectively. In contrast, at limit 4, Osiris-Plus has only 2.6% write traffic overhead. With larger limit values, e.g., 32, Osiris-Plus has nearly zero extra writes, whereas Osiris has write-traffic overhead similar to WB. Based on this observation, we use limit 4 as a reasonable trade-off between the performance overhead and the required additional stages or extra checking at the time of recovery.

Impact of Osiris and Osiris Plus persistency on multiple benchmarks

As we now understand the overall impact of Osiris-limit, i.e., N, on performance and write-traffic, we now zoom-in on the performance and write-traffic of individual benchmarks when using limit 4 as suggested in Section 2.

As we can observe, for most of the benchmarks, Osiris-Plus outperforms all other schemes in both performance and reduction in number of writes (Figure 2.9 and Figure 2.10). Meanwhile, for most benchmarks, Osiris performs close to WB scheme. As noted earlier, Osiris performance and write-traffic are bounded by the WB scheme; if the updated counters rarely get persisted before eviction from counter cache, i.e., updated less than N time, then Osiris performs similar to WB but without need for battery. Note that it is not common to have the same counter updated many times before eviction from counter cache; once a data cache block gets evicted from the cache hierarchy, it is very unlikely that it will be evicted again very soon. However, since Osiris-Plus can additionally eliminate premature (before Nth write) counter evictions, it can actually outperform WB. The only exception we can observe is `Libquantum`, which is mainly due to its behavior of repeated streaming behavior over a small array (4MB array); many cache hierarchy evictions due to conflicts, however, since each counter cache block covers the counters of 4KB page, many evictions/writes of the same data block will result in hits in the counter cache. Accordingly, a WB scheme performs better as there are few evictions (write-backs) from counter cache compared

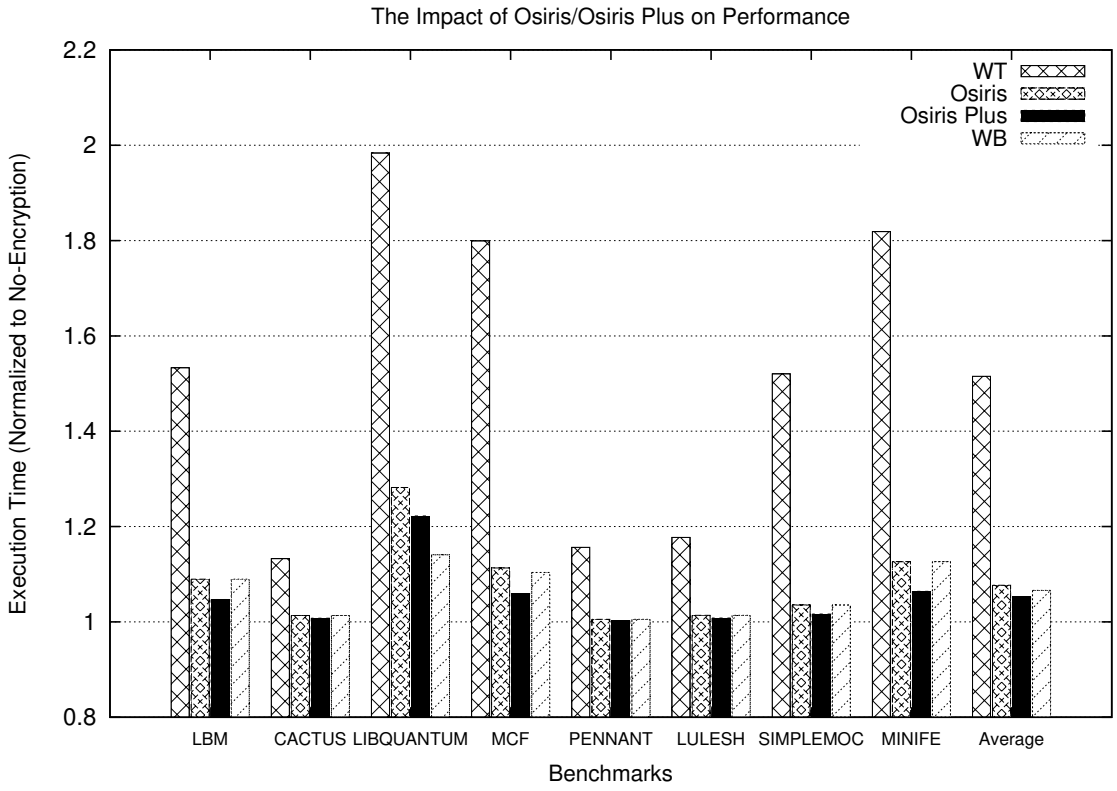

Figure 2.9: The impact of Osiris and Osiris Plus persistence on performance.

to the writes due to persistence of counter values at each Nth write in Osiris and Osiris-Plus. However, we observe that with larger limit value, such as 16, Osiris-Plus clearly outperforms WB (7.8% vs. 14% overhead), whereas Osiris performs similarly. In summary, for all benchmarks, Osiris and Osiris-Plus performs better than strict-counter persistence (WT) and very close or even better than battery-backed WB scheme. In addition, we also tested some aggressive cases, such as read latency 300ns and write latency 1000ns. On average, the Osiris-Plus outperforms WB in both execution overhead and number of writes; Osiris, although slightly degraded, is still very close to WB scheme's performance.

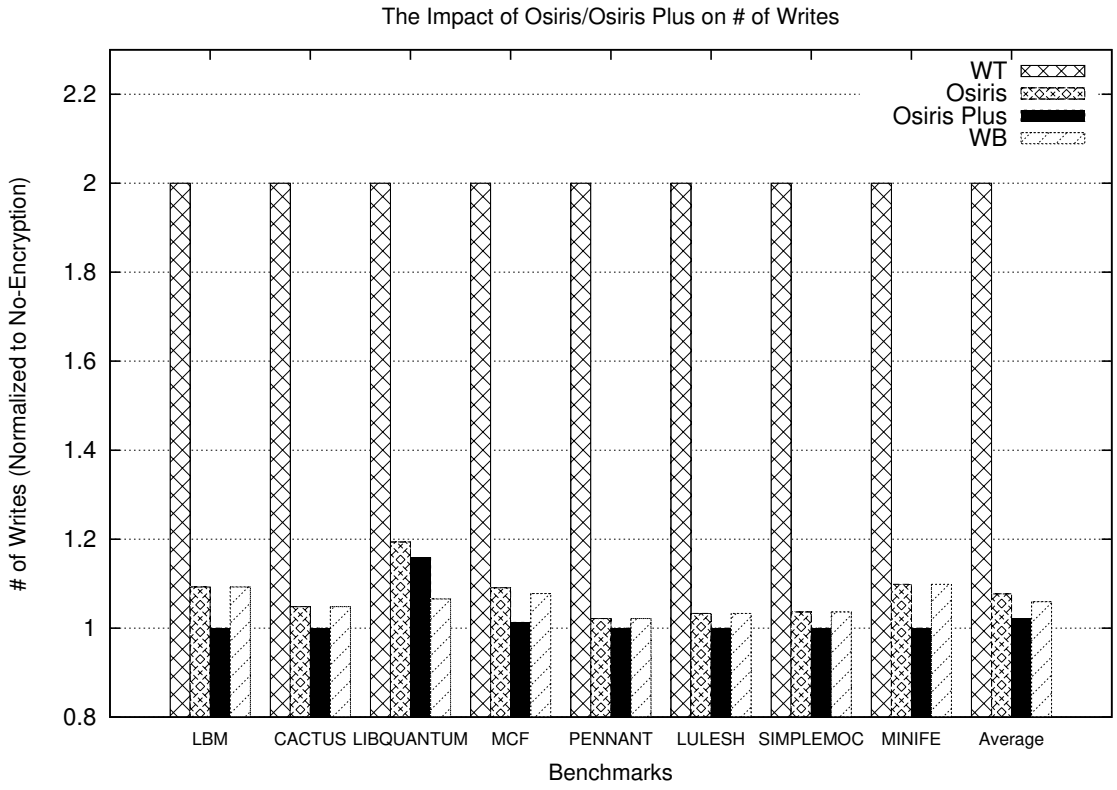

Figure 2.10: The impact of Osiris and Osiris Plus persistence on number of writes.

CHAPTER 3: OSIRIS GLOBAL

Background

As discussed above, for memory encryption, direct encryption and counter-mode encryption are the mainstream approaches. However, counter-mode encryption is commonly used in secure processor settings due to its performance advantages over direct encryption. The key step is that counter-mode encryption overlaps the generation of one-time pad through a counter-based IV with the data fetching process. This, taken together, reduces the decryption latency. In counter-mode encryption, various counter organizations are used and have distinct structures, as described in Table3.1. In state-of-the-art designs [38], each IV consists of a unique ID of a page (to distinguish between swap space and main memory space), page offset (to guarantee different blocks in a page will get different IVs), a per-block *minor* counter (to make the same value encrypted differently when written again to the same address), and a per-page *major* counter (to guarantee the uniqueness of IV when minor counters overflow).

In contrast, recent works leverage local counter and global (monolithic) counters [44, 39, 73]. In such schemes, a global(monolithic) counter can be thought of as a timer, incremented on each memory write, and due to its mono-increasing property, it can be used to generate the OTP of each block to be written to memory. The counter value will be saved in the memory along with the written block and used later for decryption, whereas a local counter is incremented only when its associated block is written back. During encryption and decryption, the local counter value is appended to the local data address to make the IV unique. Clearly, such implementations are simple to implement, but require high storage overhead: 64-bit counter value for each data block.

Table 3.1: Counter Types Used in Encryption Mode

Type	Structure	Incremental Policy	Overflow
Global (Monolithic)	Only one global counter	For every write-back to a data block, the global counter adds one	Frequent
Local	Each data block has one local counter	For every write-back to a data block, this block's local counter adds one.	Not frequent
Split	Each data block has a local major counter(per page) and a local minor counter(per block)	For every write-back to a block, its minor counter adds one. Every minor counter overflows, major counter adds one and all the data blocks in that page have to re-encrypt using new major and minor counter combination.	Not frequent

Design

As mentioned earlier, encryption counters can be organized in different ways: global, local or split counters (Table 3.1). The global (monolithic) counter referred to here is described in [38] and is similar to the central counter mentioned in connection with secure enclave[41], which is incremented by one for every writes to the main memory. On the other hand, local counters are per-block counters that are incremented whenever its associated memory block is written. Finally, in the split counter scheme, subsequent encryption of the same block would just increment its split-counter value (7-bit minor counter) by one.

As discussed previously in Chapter II, *Osiris* employs stop-loss mechanism in consecutive increment of per-block counters. However it cannot be directly applied to global/monolithic counter scheme where only one global counter records all the writes to the data blocks. In global/monolithic counter scheme, each write will increment the global counter, and the corresponding written block will copy the value of the global counter as its own counter value. Hence the subsequent updates to a specific block might lead to a large increment of the associated global counter value. This is due to the fact that, many encrypted writes to other blocks are possibly performed in between these two

subsequent writes. Consequently, employing *Osiris* in monolithic counter scheme is challenging as the counter value for a specific block does not increment consecutively between consecutive block updates, upon which *Osiris*'s recovery is based. It is impractical to try out all the possible values less than or equivalent to the global counter value to match the Merkle Tree root, since this immense number of trials could be multiplied by thousands of lost blocks (due to crash).

Even though Osiris can not be directly applied to global counter scheme, we still can use a method that reflects Osiris's spirit to solve this problem. Similar to phase number N in Osiris, we will take advantage of a much larger *epoch number* (EN), compared to the N value used in Osiris. The goal of this design is to guarantee that for each stale counter in memory we can find its up-to-date counter value within an interval of [Global Counter Value - EN, Global Counter Value]. To achieve that, we want to persist counters in cache after every EN writes.

In a naive implementation, whenever a result of the global counter modulo EN is equal to zero, all the cache lines in the counter cache could be persisted in main memory. As such, if a crash happens, the lost counter values in the counter cache will be at most EN away from those that have been persisted in main memory. To recover the lost counters, we can again use ECC bits to identify correct counters from all possible counter values as depicted in *Osiris* scheme (Chapter I Design Section). However, this process definitely demands more recovery time compared to *Osiris* implementation, since the value of EN is significantly larger than the phase number N used in general *Osiris* scheme. Apart from demanding longer recovery time, this naive implementation can also slow down or stall the system due to the frequent flushing of entire cache. Note that EN value is negatively correlated with the frequency to persist cache lines to main memory. If the value of EN is chosen to be small, recovery time could be reduced at the cost of higher frequency to persist cached counters. Therefore, the EN value must be chosen carefully. We expect it to be a relatively large value, in hundreds, or thousands to reduce the persistence overhead.

To ensure low overheads, we leverage a scheme that endeavors to reduce the redundant persistence of counters as much as possible and guarantees the lost counters will be within the range of [Global Counter-EN, Global Counter]. To this end, we require a volatile hardware-based Epoch Reference Table (ERT) with EN entries, i.e., 1024 as shown in the Figure 3.1. It has 8KB for 8 byte counter addresses and 8KB for 8 byte timestamp, to track the history of the past EN writes. Additionally, we need to add a column in counter cache: the timestamp (64-bit) (Figure. 3.1). Epoch Reference Table serves to record the counter value address and the counter updated time for every memory write. On each write in the counter cache, we consult the table to find out which counter address was written EN writes ago by checking the entry index of (Global Counter Value % Epoch Number), and check whether the counter value of that old address needs persistence in NVM. If yes, we persist the counter value to main memory and update the timestamp of the corresponding cacheline in cache. Finally, we need to update the entry of current index with the latest counter write address and write/persistent timestamp.

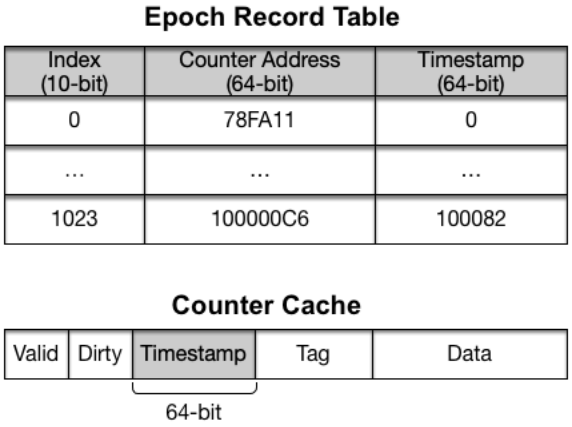

Figure 3.1: The structure of the Epoch Record Table (ERT) and the counter cache

The key idea here is that, to determine whether to persist a counter relies on the comparison result of timestamps for that counter both in the ERT and in the counter cache. It should be noted that, if the checked counter cannot be found in the cache, or it is a hit without dirty-bit set, it must have been evicted to the memory within last EN writes. Hence no further efforts need to be done in

these cases.

The timestamps in the cache either indicates the latest update time or the persistence time to main memory. The timestamp in Epoch Reference Table always reflects the timestamp the new entry is recorded. Note that, for all the newly inserted cache lines, their timestamps in cache are set to 0. Thus if a timestamp of an old entry in the ERT is larger than the timestamp of the cache line (assuming that it is found in the cache) in condition that cache line has the dirty bit set, it indicates the cache line has not been persisted within recent EN updates. Therefore, we have to persist it before evicting the old entry. Similarly, if a checked cache line has a later(larger) timestamp than that from a table entry, it means the counter in cache has been persisted recently and hence needs no redundant persistence again. That is to say the checked table entry can be safely evicted for a new counter update content. The detailed scheme of Epoch Reference Table update and counter persistency is displayed in the algorithm 1. To help the readers to understand the whole picture of the algorithm, we also draw a flowchart 3.2 to diagram the algorithm.

When using this scheme, our table behaves like a circular buffer that helps us know which counter block was updated EN writes ago. We need flush the block if it exists in the cache and is dirty and has not been recently persisted. By doing so, we can ensure that any updated counter block, if gets lost due to a crash, has a value between $[GloablCounter - EpochNumber, GlobalCounter]$.

Write Operation of Osiris-global

The write operation for Osiris-global is shown in Figure 3.3. The distinguished step in write operation is shown in ③. After updating the latest counter for the coming write request in cache, we need to find the entry index in Epoch Reference Table to record this update. Before updating an entry in the ERT, we will check the old entry in cache. There are two cases in which we proceed directly to steps ⑥ and ⑦ after overwriting the old counter address and timestamp in the table

Algorithm 1: Epoch Reference Table Update and Counter Persistency

Data: Counter Cache(CC), Epoch Reference Table(ERT), Epoch Number(EN), Global Counter(GC)

Result: Update ERT for each counter write, and persist counter recorded in ERT EN writes ago conditionally

```
Initialization;
 Timestamp in all cachelines is set to 0;
 Timestamp in any inserted cachelines is set to 0;
 All colunms in ERT are set to 0;
 GC = 0;
while MC gets a write request do
    Update counter in cacheline;
    TimePoint CurrentTime = current time;
    GC++;
    Index i = GC%EN;
    if ERT[i].Address in CC and Dirty then
        Tag T=ERT[i].Address/Data blocks' size for a cacheline;
        if CC[T].Timestamp ≥ ERT[i].Timestamp then
            ERT[i].Address = Counter address with GC value;
            ERT[i].Timestamp = CurrentTime;
        else
            Write CC[T] to main memory;
            CurrentTime = current time;
            CC[T].Timestamp = CurrentTime;
            CC[T].Dirty = False;
            ERT[i].Address = Counter address with GC value;
            ERT[i].Timestamp = CurrentTime;
        end
    else
        ERT[i].Address = GCth write's counter address;
        ERT[i].Timestamp = CurrentTime;
    end
end
```

with the latest write information. First, if the address in an entry is present in cache, dirty, and its timestamp in the table is larger than the timestamp in cache. Second, the counter address of an entry is either not in cache or not dirty. Otherwise, we have to persist the counter in the cache to NVMM before moving on to the subsequent steps.

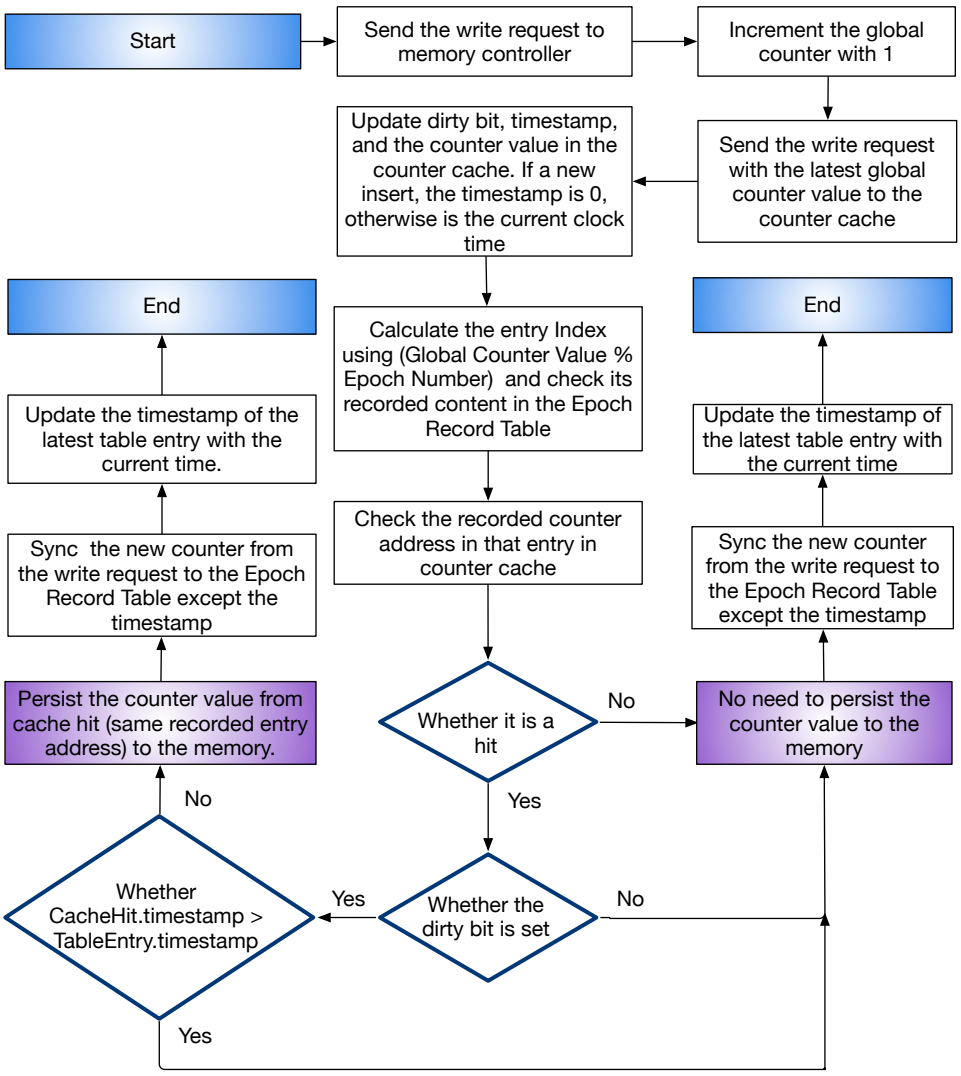

Figure 3.2: The flowchart of Epoch Record Table (ERT) update and counter cache update

Hardware Overhead for Osiris-global

For Osiris-global, the hardware complexity involves a hardware table(Epoch Record Table) and a timestamp column in the cache. The hardware table has a size less than 18K and the size of timestamp column of 64-bit roughly introduces less than 12.5% additional counter cache size. Both are within an acceptable overhead range.

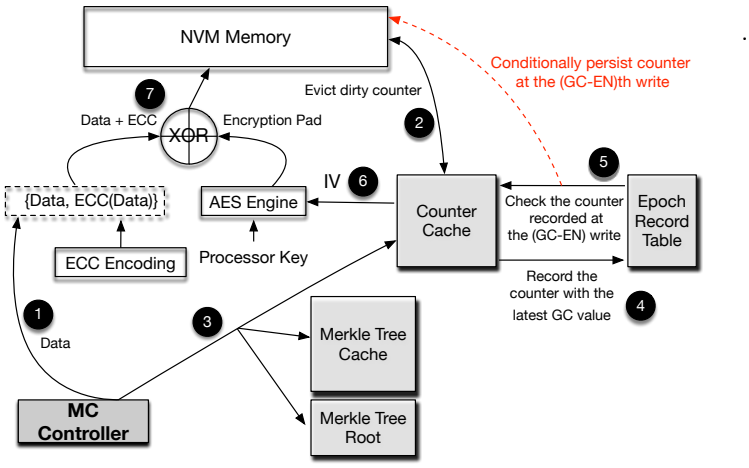

Figure 3.3: Osiris-global write operation

Evaluation

The impact of Epoch Number

Recall that in Osiris-global, we introduce an epoch number(EN). Accordingly, before we record updated counter address present in counter cache to an entry of Epoch Reference Table, we need to check whether to persist a counter address written EN times ago in that specific entry. First, we did a sensitivity study to determine the appropriate EN to use in our experiments and the result is shown in Figure 3.4. The comparison demonstrates that the larger the EN is, the impact on performance is more close to WB scheme and with a longer recovery time because of too many trials one has to attempt to identify a data-and-ECC match. As such, we pick 1024 as our Epoch Number.

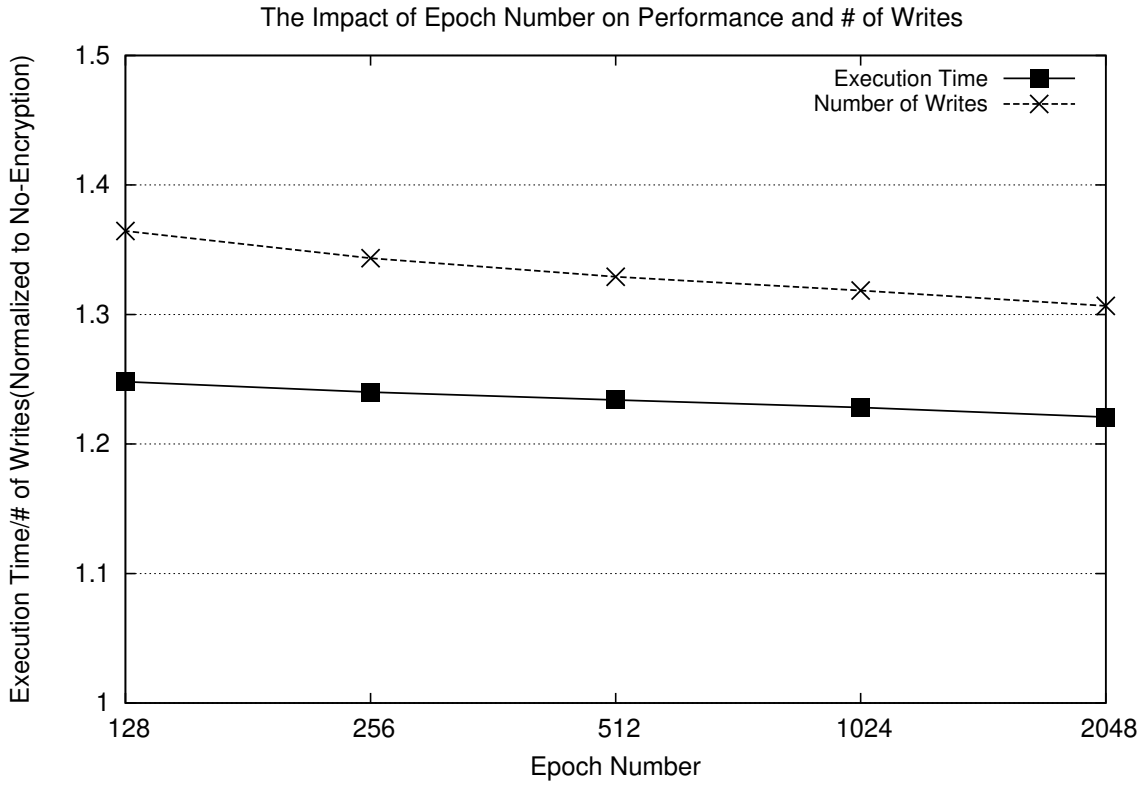

Figure 3.4: The impact of Epoch Number on performance (Number of writes).

Osiris-global persistency on multiple benchmarks

As we did for Osiris/Osiris Plus, we conducted the sensitivity study for Osiris-global. Since global counter compared to split counters does not provide good spacial coverage, hence the overall counter cache miss rate is high, even at the size of 4M(11.4% miss rate) in Figure 3.5. It is admitted that the miss rate decreases with a large cache size but such performance improvement is not very significant. When increasing the cache size from 256K to 4M, it only reduces the miss rate by 6.5% . Considering a cache takes the major space of the processor chip and requires large amount of transistors and energy, we traded off the miss rate gains for small size of 256K as the cache size for Osiris-global.

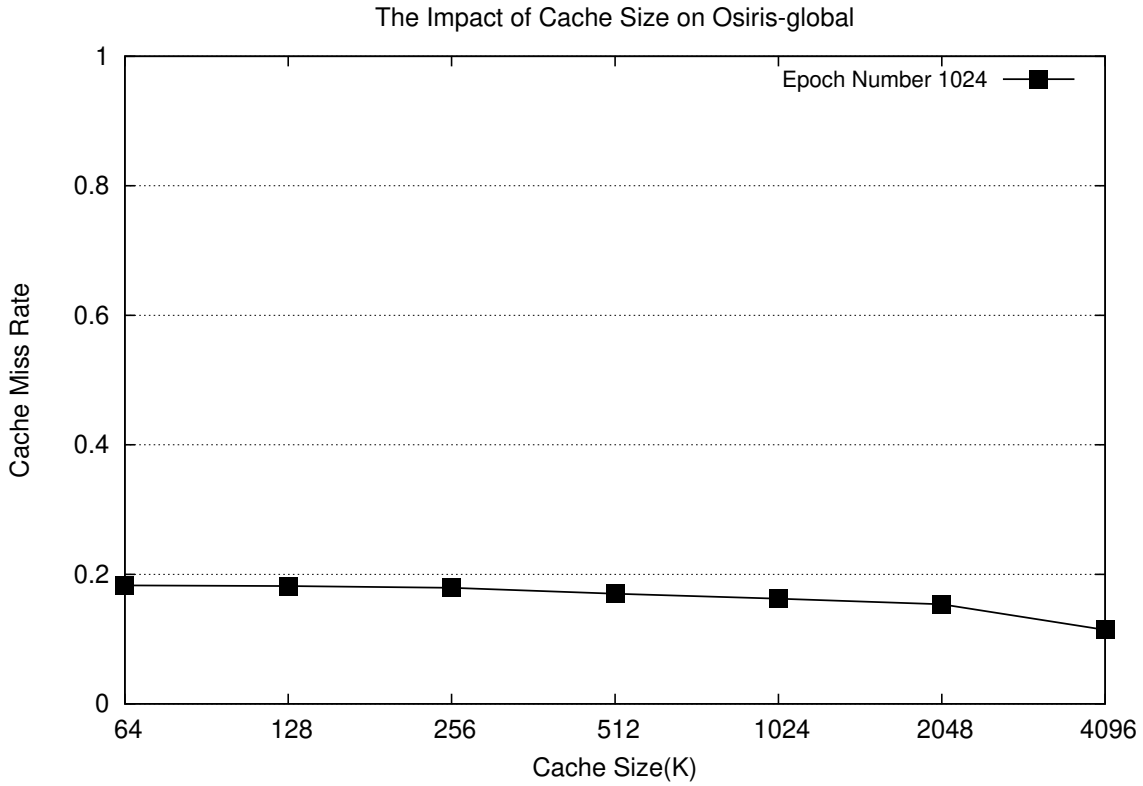

Figure 3.5: The sensitivity studies of cache size for Osiris-global

Our results show that using EN of 1024, the overhead of Osiris-global on average is comparable to that of Osiris/Osiris Plus in Figure 3.6. On average, the execution time overhead is an extra 3.88% to WB scheme. For some application, such as `Libquantum`, the percentage of overhead normalized to WB scheme could reach as high as 10%, which is consistent with its performance under Osiris scheme. For computation-intensive graph algorithms, their behavior on performance are similar to the memory-intensive application except TSP and we know it is because TSP is a no-write application. Meanwhile, the average number of writes under Osiris-global scheme causes additional 17.2% overhead in comparison to WB scheme and is 62.4% lower than WT scheme in Figure 3.7. Besides, `Pennant`, `Lulesh` and `Simplemoc` contribute the most write overhead, all above 20%. This phenomenon is due to temporal write correlation pattern of application up-

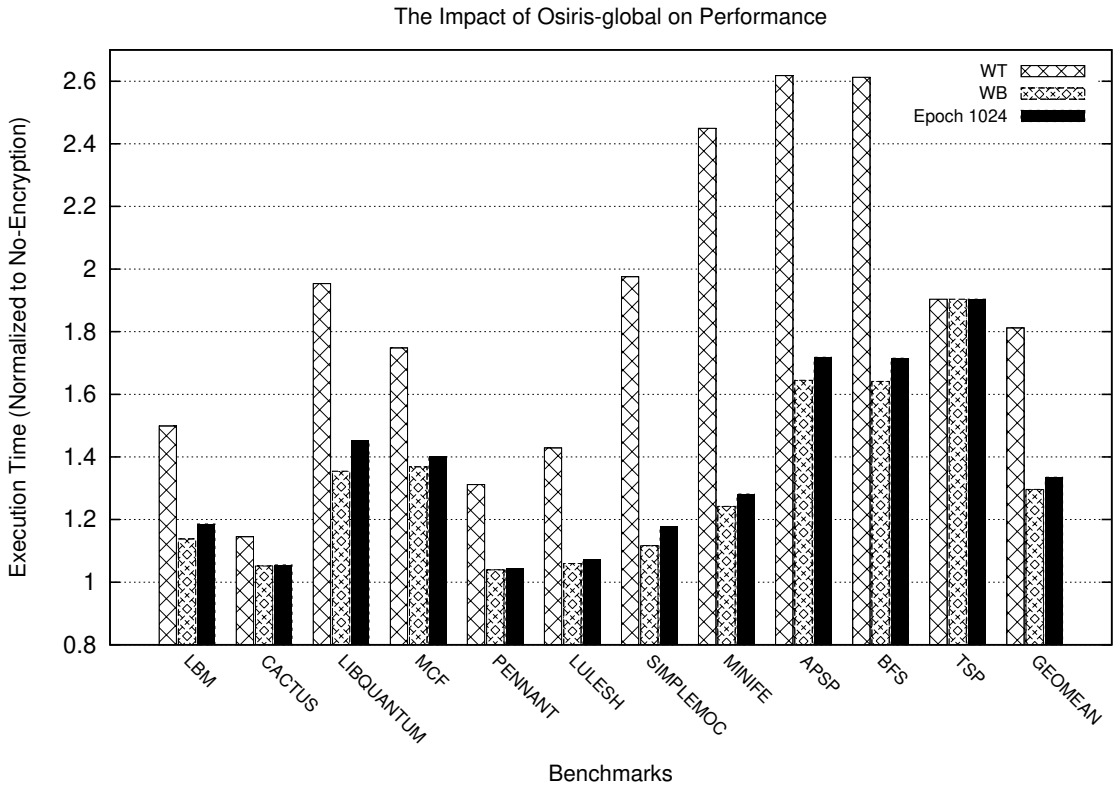

Figure 3.6: The impact of Osiris-global on performance.

date. For graph algorithms, we observe a closing gap between WT and WB in terms of number of writes. This is because graph algorithms have a higher miss rates than their memory-intensive counterparts (3-folder higher), which leads to frequent write-backs. In summary, the results obtained for performance and number of writes under Osiris-global matches our expectation, that is, to outperform WT scheme but is worse than WB scheme in an acceptable range.

Recovery Time

Similarly, for the monolithic counter scheme, the additional recovery time is bounded by $131072 \times 1024 \times 100ns$, which is roughly 13.1 seconds. However, note that the overhead is constant and

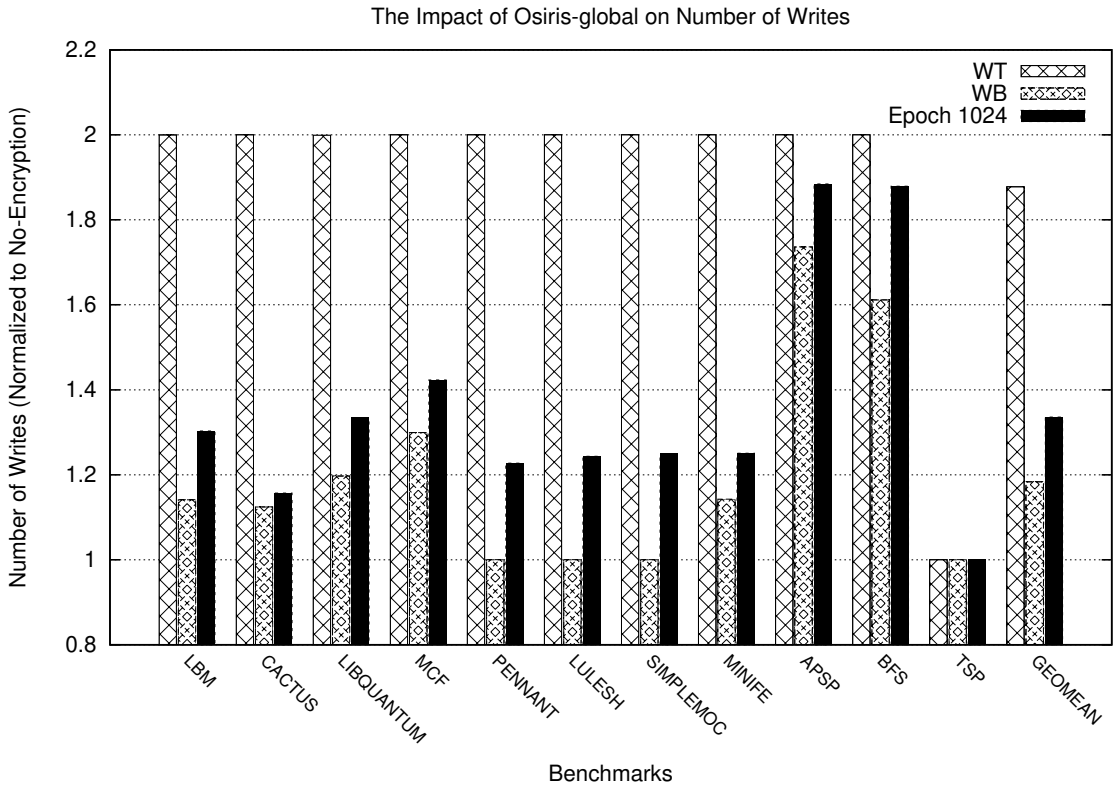

Figure 3.7: The impact of Osiris-global on number of writes.

only affected by the cache size, thus for large memory sizes the overhead would be marginal. For instance, if we use 1TB memory, the recovery time overhead will be less than 1%, and becomes much smaller for larger memories. After a second thought, since we are unlikely to persist counters within the range of [Global Counter-Epoch Number, Global Counter] after a crash event, it also means there are Epoch Number of stale counters present in memory at most if not considering other uncorrectable errors on counters. So the recovery time can be reduced to $1024 \times 1024 \times 100ns$, which is only 0.102 seconds. Although we can take use of multiple AES engines for Osiris-global for recovery, it is not necessary because Osiris-global does not affect regular read/write operation like Osiris plus does.

CHAPTER 4: SUMMARY

In this research, I have studied the crash recovery problem in secure NVM, especially for the PCM. Unlike regular NVM, secure NVM has security metadata that can be updated in the NVM as frequently as data blocks are, because they need to be persisted in addition to data blocks. Otherwise, if a system crash occurs, the inconsistency between metadata and their corresponding data blocks would lead to the failure in data recovery.

Therefore, I explored several metadata persisting schemes, evaluated their performance and compared them with existing persistent schemes. Our novel schemes mainly take two steps. First, the metadata, say, the counters will be persisted under the condition that their values modulo phase-change number equals to 0. As such, the number of writes that would shorten the life-span of PCM will be reduced significantly. Second, the ECC bits for data blocks would be generated pre-encryption instead of post-encryption so that they can function as a sanity-check guardian to identify the latest counters if a crash takes place. Meantime these ECC bits still can correct and detect errors. We call these schemes Osiris and Osiris Plus, with the latter being a more aggressive version of the former to enhance the performance by evicting premature counters. Both schemes function similarly to the strict counter persistence scheme but significantly reduce the performance overhead and NVM writes. Then I discussed how Osiris can work and be integrated with state-of-the-art data and counter integrity verification (ECC and Merkle tree) for rapid failure/counter recovery. I used these schemes to compare with other memory persistency schemes in respective of trade-off between hardware complexity and performance. The evaluation in eight representative benchmarks shows desirable performance overhead and NVM writes via employing Osiris/Osiris Plus in comparison with other schemes discussed.

A further exploration in a different memory counter-mode encryption implementation brings in another persistent scheme. For that work, we proposed an epoch-based persistent scheme for a global counter. In that scheme, the number of counter in need to be recovered will be limited by the epoch size. The number of trials for identifying the correct counter will also be limited by the epoch size. This scheme introduces a low-overhead hardware table but incurs a small number of additional writes.

In summary, this research enables the recovery of secure NVM systems and hence assists applications to take advantage the persistency of NVMs without programming burden. Meanwhile, it helps to saves NVM's lifespan by minimizing the number of writes required to achieve such crash consistency requirement for secure NVM systems.

LIST OF REFERENCES

[1] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption." http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD-Memory-Encryption-Whitepaper-v7-Public.pdf, 2016.

[2] M. Henson and S. Taylor, "Memory encryption: A survey of existing techniques," *ACM Comput. Surv.*, vol. 46, pp. 53:1–53:26, Mar. 2014.

[3] V. Costan and S. Devadas, "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[4] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "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*, HASP 2016, (New York, NY, USA), pp. 10:1–10:9, ACM, 2016.

[5] J. Salter, "Intel promises full memory encryption in upcoming cpus." https://arstechnica.com/gadgets/2020/02/intel-promises-full-memory-encryption-in-upcoming-cpus/, 2020-2-26.

[6] Micron, "Breakthrough nonvolatile memory technology." https://www.micron.com/products/advanced-solutions/3d-xpoint-technology, 2016-08-10.

[7] "Intel Optane DC Persistent Memory." https://www.intel.com/content/www/us/en/...and.../optane-dc-persistent-memory.html, 2018.

[8] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE micro*, vol. 30, no. 1, pp. 143–143, 2010.

[9] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," Feb 2018.

[10] P. Chi, S. Li, Yuanqing Cheng, Yu Lu, S. H. Kang, and Y. Xie, "Architecture design with stt-ram: Opportunities and challenges," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 109–114, Jan 2016.

[11] I. Alam, S. Pal, and P. Gupta, "Compression with multi-ecc: enhanced error resiliency for magnetic memories," in *Proceedings of the International Symposium on Memory Systems*, pp. 85–100, 2019.

[12] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, *et al.*, "Phase change memory technology," *Journal of Vacuum Science & Technology B, Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena*, vol. 28, no. 2, pp. 223–262, 2010.

[13] H. Wang and X. Yan, "Overview of resistive random access memory (rram): Materials, filament mechanisms, performance optimization, and prospects," *physica status solidi (RRL)–Rapid Research Letters*, vol. 13, no. 9, p. 1900073, 2019.

[14] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating stt-ram as an energy-efficient main memory alternative," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 256–267, IEEE, 2013.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 2–13, 2009.

[16] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 1537–1550, May 2016.

[17] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, (Santa Clara, CA), pp. 323–338, USENIX Association, 2016.

[18] "Windows® Server 2016/NVDIMM-N Solution - Micron Technology, Inc.." https://www.micron.com/-/.../hpe-micron-microsoft-windows-server-2016-nvdimm-..., 2016.

[19] "Direct access for files." `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`. Accessed: 2014.

[20] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, May 2015.

[21] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, (New York, NY, USA), pp. 3–18, ACM, 2015.

[22] V. Young, P. J. Nair, and M. K. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," in *Proceedings of the Twentieth International Conference on Architectural Sup-*

port for Programming Languages and Operating Systems, ASPLOS '15, (New York, NY, USA), pp. 33–44, ACM, 2015.

[23] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 14–23, IEEE, 2009.

[24] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 263–276, ACM, 2016.

[25] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 383–394, 2010.

[26] J. Yue and Y. Zhu, "Accelerating write by exploiting pcm asymmetries," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 282–293, IEEE, 2013.

[27] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "Improving write operations in mlc phase change memory," in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–10, IEEE, 2012.

[28] L. Jiang, Y. Zhang, B. R. Childers, and J. Yang, "Fpb: Fine-grained power budgeting to improve write throughput of multi-level cell phase change memory," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–12, IEEE Computer Society, 2012.

[29] M. Arjomand, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Boosting access parallelism to pcm-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 695–706, 2016.

[30] F. Xia, D. Jiang, J. Xiong, M. Chen, L. Zhang, and N. Sun, "Dwc: Dynamic write consolidation for phase change memory systems," in *Proceedings of the 28th ACM international conference on Supercomputing*, pp. 211–220, ACM, 2014.

[31] A. Jadidi, M. Kandemir, and C. Das, "Tolerating write disturbance errors in pcm: Experimental characterization, analysis, and mechanisms," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 53–65, IEEE, 2018.

[32] S. Cho and H. Lee, "Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 347–357, 2009.

[33] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montaño, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pp. 1–11, 2010.

[34] S. Raoux, W. Wełnic, and D. Ielmini, "Phase change materials and their application to nonvolatile memories," *Chemical Reviews*, vol. 110, no. 1, pp. 240–267, 2010. PMID: 19715293.

[35] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing dram footprint with nvm in facebook," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–13, 2018.

[36] M. Ye, C. Hughes, and A. Awad, "Osiris: A low-cost mechanism to enable restoration of secure non-volatile memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 403–415, IEEE, 2018.

[37] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *2007 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 183–196, IEEE, 2007.

[38] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 179–190, IEEE Computer Society, 2006.

[39] S. Liu, A. Kolli, J. Ren, and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 310–323, Feb 2018.

[40] P. Zuo and Y. Hua, "Secpm: a secure and persistent memory system for non-volatile memory," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, (Boston, MA), USENIX Association, 2018.

[41] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 54–62, 2016.

[42] M. Ye, K. Zubair, A. Mohaisen, and A. Awad, "Towards low-cost mechanisms to enable restoration of encrypted non-volatile memories," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[43] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 107–119, ACM, 2017.

[44] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *The 24th International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[45] R. Huang and G. E. Suh, "Ivec: off-chip memory integrity protection for both security and reliability," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 395–406, 2010.

[46] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication," 1997.

[47] M. J. Dworkin, *SP 800-38D. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. National Institute of Standards & Technology, 2007.

[48] M.-Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[49] S. Cha and H. Yoon, "Efficient implementation of single error correction and double error detection code with check bit pre-computation for memories," *JSTS: Journal of Semiconductor Technology and Science*, vol. 12, no. 4, pp. 418–425, 2012.

[50] R. Naseer and J. Draper, "Dec ecc design to improve memory reliability in sub-100nm technologies," in *15th IEEE International Conference on Electronics, Circuits and Systems, 2008. ICECS 2008.*, pp. 586–589, IEEE, 2008.

[51] R. Naseer and J. Draper, "Parallel double error correcting code design to mitigate multi-bit upsets in srams," in *Solid-State Circuits Conference, 2008. ESSCIRC 2008. 34th European*, pp. 222–225, IEEE, 2008.

[52] T. N. Miller, R. Thomas, J. Dinan, B. Adcock, and R. Teodorescu, "Parichute: Generalized turbocode-based error correction for near-threshold caches," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 351–362, IEEE, 2010.

[53] A. Awad, L. Njilla, and M. Ye, "Triad-nvm: Persistent-security for integrity-protected and encrypted non-volatile memories (nvms)," *arXiv preprint arXiv:1810.09438*, 2018.

[54] K. A. Zubair and A. Awad, "Anubis: ultra-low overhead and recovery time for secure non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 157–168, 2019.

[55] T. Instruments, "Discriminating Between Soft Errors and Hard Errors in RAM." http://www.ti.com/lit/wp/spna109/spna109.pdf, 2018-06-27.

[56] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, "Onyx: A protoype phase change memory storage array," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2011.

[57] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "Lot-ecc: Localized and tiered reliability mechanisms for commodity memory systems," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 285–296, IEEE, 2012.

[58] D. H. Yoon and M. Erez, "Virtualized and flexible ecc for main memory," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pp. 397–408, 2010.

[59] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-power, low-storage-overhead chipkill correct via multi-line error correction," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2013.

[60] S.-L. Gong, M. Rhu, J. Kim, J. Chung, and M. Erez, "Clean-ecc: High reliability ecc for adaptive granularity memory system," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 611–622, 2015.

[61] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 467–478, IEEE, 2014.

[62] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: a large-scale field study," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 193–204, 2009.

[63] X. Jian, J. Sartori, H. Duwe, and R. Kumar, "High performance, energy efficient chipkill correct memory with multidimensional parity," *IEEE Computer Architecture Letters*, vol. 12, no. 2, pp. 39–42, 2012.

[64] V. Sridharan and D. Liberty, "A study of dram failures in the field," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE, 2012.

[65] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," 1997.

[66] J. Kim, M. Sullivan, and M. Erez, "Bamboo ecc: Strong, safe, and flexible codes for reliable computer memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 101–112, IEEE, 2015.

[67] X. Jian and R. Kumar, "Adaptive reliability chipkill correct (arcc)," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 270–281, IEEE, 2013.

[68] S. Wang, F. Wu, Z. Lu, Y. Zhou, Q. Xiong, M. Zhang, and C. Xie, "Lifetime adaptive ecc in nand flash page management," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1253–1556, IEEE, 2017.

[69] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[70] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *FIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

[71] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[72] M. Heroux, R. Neely, and S. Swaminarayan, "Asc co-design proxy app strategy." `http://www.lanl.gov/projects/codesign/proxy-apps/assets/docs/proxyapps-strategy.pdf`, 2013-01-06.

[73] S. Gueron, "A memory encryption engine suitable for general purpose processors." https://eprint.iacr.org/2016/204, 2016.