ABSTRACT

HAN, XIJING. Efficient Memory Security Support Under Realistic System Constraints. (Under the direction of James Tuck and Amro Awad).

Secure memory offers two critical data protection measures: data confidentiality and data integrity, which are important to provide trusted computing environment. Deploying secure memory in computer systems incurs performance overhead and challenges. This dissertation considers the challenges of adopting secure memory in persistent memory systems and embedded systems. The first three works propose solutions to solve the performance problems faced by secure persistent memory. The last work proposes solutions to solve the challenge of providing data confidentiality in embedded systems with strict hardware resource constraints.

The first work explores run-time performance overhead of running persistent applications on conventional secure processor architectures. Persistent applications use long-latency flush instructions and memory fences to make sure that writes to persistent data reach the persistence domain in a way that is crash consistent. Intel's Asynchronous DRAM Refresh (ADR) make the on-chip Write Pending Queue (WPQ) part of the persistence domain and help reduce the penalty of persisting data since data only needs to reach the on-chip WPQ to be considered persistent. However, when persistent applications run on secure processors, for the sake of securing memory many cycles are added to the critical path of their write operations before they ever reach the persistent WPQ, preventing them from fully exploiting the performance advantages of the persistent WPQ. Dolos is proposed to make it feasible for secure persistent applications to benefit more from the on-chip persistence domain. Dolos prioritizes persisting data without sacrificing security by an additional minor security unit, Mi-SU, that utilizes a much faster secure process that protects only the WPQ. Thus, the secure operation latency in the critical path of persist operations is reduced and hence persistent transactions can complete earlier. Dolos retains a conventional major security unit for protecting memory that occurs off the critical path after inserting secured data into the WPQ. We implemented Dolos architecture in the GEM5 simulator, and analyzed the performance of 6 benchmarks from the WHISPER suite. Dolos improves their performance by 1.66× on average.

The second work studies the significant burden on energy costs and battery size caused by securely flushing on-chip extended persistence domain (i.e., caches) upon crashes. To hide high write latency of persistent memory and simplify programming model of leveraging

persistent memory for crash consistency, recent standards advocate for sufficient back-up power that can flush the whole cache hierarchy to the persistent memory upon detection of an outage, i.e., extending the persistence domain to include the cache hierarchy. In the secure NVM with extended persistence domain (EPD), in addition to flushing the cache hierarchy, extra actions need to be taken to protect the flushed cache data. We demonstrate that naive implementations could lead to significantly expanding the required power hold-up budget (e.g., 10.3× more operations than EPD system without secure memory support). The significant overhead is caused by memory accesses of secure metadata. To reduce such overhead, we present Horus, a novel EPD-aware secure memory implementation. Horus reduces the overhead during draining period of EPD system by reducing memory accesses of secure metadata. Experiment result shows that Horus reduces the draining time by 5×, compared with the naive baseline design.

The third work observes the gap between secure persistent memory systems and future NVM interfaces, which makes the state-of-the-art solutions to reduces memory writes of secure metadata (encryption counter and message authentication code (MAC)) unsuitable for future memory interfaces. Specifically, the majority of today's solutions assume that either the encryption counter and/or MAC can be co-located with data by directly or indirectly leveraging the otherwise Error-Correcting Codes (ECC) bits. However, we observe that emerging interfaces and standards delegate the ECC calculation and management to happen inside the memory module, which makes it possible to remove extra bits for ECC in memory interfaces. Thus, all today's solutions may need to separately persist the encrypted data, its MAC, and its encryption counter upon each memory write. To mitigate this issue, we propose a novel solution, Thoth, which leverages a novel off-chip persistent partial updates combine buffer that can ensure crash consistency at the cost of a fraction of the write amplification by the state-of-the-art solutions when adapted to future interfaces. Based on our evaluation, Thoth improves the performance by an average of 1.22× (up to 1.44×) while reducing write traffic by an average of 32% (up to 40%) compared to the baseline Anubis when adapted to future interfaces.

The last work enables application-transparent data confidentiality for embedded systems without hardware memory encryption support. Some high-end embedded systems have rich hardware to support efficient and application-transparent memory encryption service. However low-end embedded systems lack such hardware support due to area and cost limitations. To enable memory encryption in such low-end embedded systems, currently, it requires programmer to carefully analyze and modify applications, which is tedious and error-prone. To fill this gap, we propose ATHENA, a novel software-based

solution to provide memory encryption for embedded systems with minimal hardware requirement. ATHENA leveraged a small on-chip memory region as trusted memory and securely transfer data between on-chip and off-chip memory through a run time library, encrypting data when data is transferred from on-chip memory to off-chip memory and decrypting data the other way around. In addition, ATHENA modifies applications to redirect off-chip memory access to on-chip memory access through a static transformation pass called Secure Memory Pass implemented in GCC compiler. We evaluate ATHENA on STM32F769I-EVAL board. Experiment results show that ATHENA reduces execution time to an average of 0.13×, compared to an intuitive design without re-using decrypted data.

Efficient Memory Security Support Under Realistic System Constraints

by
Xijing Han

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2024

APPROVED BY:

Aydin Aysu

Man-Ki Yoon

James Tuck
Co-chair of Advisory Committee

Amro Awad
Co-chair of Advisory Committee

BIOGRAPHY

Xijing Han was born in Shanghai, China in 1989 and spent most of her life there before college. In 2008, Xijing went to Southeast University, Nanjing, China and received her Bachelor Degree in 2012. She received Master Degree from Brown University, Rhode Island, USA in 2016. In 2019, she started her Ph.D. at North Carolina State University. Her research work focuses on memory security issues for persistent memory systems and embedded systems.

ACKNOWLEDGEMENTS

First, I'd like to thank my Ph.D. advisors, Dr. James Tuck and Dr. Amro Awad. It is their guidance and suggestion that helps me on the path to achieve the Ph.D. Degree.

Second, I'd like to thank the members of my PhD advisory committee, Dr. Aydin Aysu and Dr. Man-Ki Yoon. Thank you for your valuable feedback and attending my final exam.

Last, I'd like to thank my mother and father, Xianlin Ge and Bin Han. They are the most reliable source of support during the study of my Ph.D.. I want to extend my thanks to my lovely grandmother, for teaching me so many valuable life lessons.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

With security becoming a first-class citizen design requirement for computing systems, most processor vendors are racing into providing security primitives that enable a safe execution environment(5; 8; 31; 2; 20; 36). While the threat model and hence security guarantees, implementation details, and maturity of these supports vary significantly, a common theme between all of them is the need for protecting data confidentiality and integrity when leaving the processor chip. While efficient memory integrity and confidentiality protection have reached to an acceptable maturity level for server class systems with conventional memory (i.e., DRAM) (47; 56; 26), adopting memory integrity and confidentiality still faces challenges for computer systems with resource constraints. In this dissertation, we will discuss several challenges of adopting secure memory under different system resources constraints and their solutions

1.1 Thesis Statement

In this dissertation, we investigate the challenges of providing memory security under following system limitations.

(1) Limited on-chip battery power/size

Persistent memory has the ability like storage memory to retain data after power outage or crash events with much faster data access latency than storage memory, which makes it a tempting replacement as main memory. In persistent memory systems, part of on-chip memory resource can be made persistent with battery power support to flush their content to persistent memory upon crash. Such on-chip "persistent" memory includes a small write buffer in memory controller called Write Pending Queue (WPQ) and with larger battery power, cache hierarchy can also be included. Conventional memory secure operation is essentially designed to protect off-chip memory and inefficient to secure on-chip "persistent" memory upon crash since they consume significant battery power. On the other hand, using conventional memory secure operation towards on-chip WPQ or cache during run-time incurs significant performance overhead since these operations are expensive. In this dissertation, we propose two works, Dolos and Horus, to efficiently secure on-chip "persistent" WPQ and cache respectively with minimally increasing battery power budget.

(2) Absence of extra bits in memory interface

To provide memory security for persistent memory systems, crash consistency of both data and secure metadata (counters, MACs, MT nodes) should be guaranteed. To ensure crash consistency of secure NVMs while incurring minimal write traffic and performance overheads, state-of-the-art solutions(60; 65; 19; 58; 27; 59) rely on persisting (part of) the security metadata atomically with data through repurposing Error-Correcting Codes (ECCs) that are co-located with the data. In all these works, the majority of the write reduction is based on leveraging the ECC bits (typically 64-bits) written atomically along with the data. However, with emerging on-die ECC module, ECC can be generated by memory module internally instead of relying on processor, hence extra ECC bits may no longer be needed in future memory interface, which make all prior works leveraging ECC bits to persist security metadata ineffective. To solve this chanllange, in this disseration, we propose Thoth to provide recoverability for secure metadata without adding significant write overhead in systems with no ECC bits in memory interface.

(3) Absence of dedicated hardware support for memory encryption

Today's secure memory solutions(5; 2) are mainly implemented for server systems mostly targeting confidential computing in cloud systems(31; 9; 22) and assume the existence of dedicated hardware for memory security. With attackers' physical access become even more plausible for IoT devices, at least similar guarantees should be provided. However, in some legacy embedded systems and systems with limited power/area budget, such hardware support does not exist. Importantly, there is an increasing compliance requirements and regulations around protecting users' data which necessitates exploring solutions that can be applied without hardware changes or system replacement. To enable data confidentiality with minimal programmer efforts for embedded systems with limited hardware resources, in this dissertation, we present the first academic work, ATHENA, that discusses the challenges of a software-based memory encryption scheme, the design space, and presents a compiler-based software-based memory encryption solution with minimal to no programmer intervention.

1.2 Thesis Contributions

In this section, we discuss the contributions of this dissertation by introducing how we solve the challenges raised in previous section.

Dolos: This work addresses part of the first challenges of securing on-chip persistent WPQ without adding on-chip battery power budget. Dolos proposes a lightweight run-time secure operation before persistency (i.e., WPQ). WPQ is secured during run-time therefore its content can be flushed to off-chip memory immediately upon crashes without adding significant extra power budget for secure operation towards WPQ during crash time. The lightweight run-time secure operation to secure WPQ adds negligible run-time performance overhead.

Horus: This work addresses the other part of the first challenge of securing on-chip persistent cache with minimal extra power budget. Horus provides solution for low-overhead secure operation towards cache draining upon crash. Horus reduces the number of operations to secure cache upon crash in the worst case scenario. In Horus, the number of operations upon crash is proportional to the number of dirty cache lines in cache hierarchy, which makes it easy to estimate required battery power for the worst case.

Thoth: This work addresses the second challenge of reducing writes of secure metadata in future memory interface without the support of extra bits in memory interface. Thoth saves memory writes of secure metadata by combining several partial updates of secure metadata into one memory update to a reserved buffer in off-chip persistent memory and allows secure metadata update in its original memory location through natural eviction from secure metadata cache.

ATHENA: This work addresses the third challenge of providing data confidentiality without hardware memory encryption module for embedded systems. ATHENA provides transparent software solution of memory encryption with the support of compiler, a run-time software library and hardware resources that are typical/already existed in embedded systems.

1.3 Thesis Organization

The rest of this dissertation is organized as follows. In Chapter 2, we discuss the background topics related to this dissertation. In Chapter 3, we introduce our Dolos work on efficiently securing on-chip WPQ in ADR-supported persistent memory systems without incurring significant power overhead. In Chapter 4, we introduce our Horus work which securely flushes on-chip cache upon crash in persistent memory systems with extended persistent domain (e.g., cache) with low power overhead. In Chapter 5, we discuss our Thoth work, which reduces memory writes of persisting secure metadata in future memory interface which may not have extra bits to be re-purposed with secure metadata information. In Chapter 6, we discuss our ATHENA work on providing flexible software solution of data confidentiality for resource limited embedded systems without hardware support of memory encryption.

CHAPTER

2 BACKGROUND

2.1 Secure Memory

In most trusted execution environments, e.g., Intel's Software Guard Extension (SGX)(31), two critical data protections can be offered: (1) ***Data Confidentiality***, and (2) ***Data Integrity***. Both integrity and confidentiality of data have been emphasized as necessary protection measures for secure execution environments. Below, we briefly describe how these two primitives are commonly implemented.

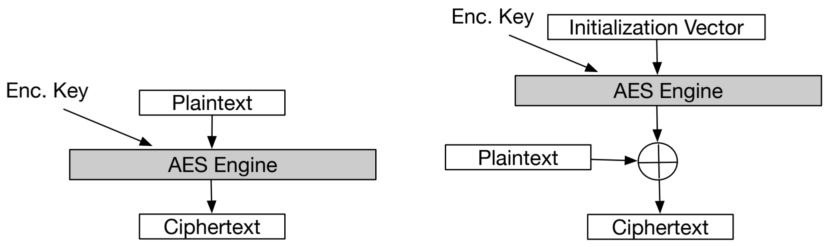
2.1.1 Memory Encryption

With attacks that leverage data remanence in memory devices, e.g., cold boot attacks in DRAM, there have been significant efforts to encrypt memory with low overheads. Such attacks become even more plausible with the use of emerging NVMs that naturally retain the data for a long time after a power outage. Generally, memory encryption can be implemented either inside the memory module or from the processor side (near the memory controller). In the former approach, e.g., as in i-NVMM(21), the data is encrypted/decrypted when written/read inside the memory module. Thus, any physical attacks that could successfully acquire the memory module would fail to breach the confidentiality of data. However, malicious implants or hardware trojans in the memory bus or memory slots can observe the memory data not encrypted and hence breach confidentiality. Due to the challenges of ensuring a trustworthy supply chain of integrated circuits (including motherboards), there has been strong traction towards limiting the trust base to processor chips, and hence moving memory encryption to the memory controller on the processor side. For instance, Intel's Software Guard Extension (SGX) and AMD's Secure Memory Encryption (SME) implement memory encryption on the processor side. By doing so, any attempts to breach the data confidentiality outside the processor chip will be thwarted via encryption. For the rest of this dissertation, we will assume processor-side memory encryption due to its prevalence and stronger security.

There are two major ways to implement processor-side memory encryption: **direct encryption** or **counter-mode encryption**.

Direct Encryption: As shown in Figure 2.1-a, the plaintext is used as a direct input for the cryptographic encryption engine (e.g., AES) to generate the ciphertext. In addition to the plaintext, a processor-wide (or per enclave in the case of SGX) encryption key is used as the second input to the encryption engine. In this scheme, the same plaintext will always generate the same ciphertext, which opens the door for dictionary-based attacks. While the memory address can be augmented with the plaintext to minimize the dictionary-based attacks, temporal reuse of the same value in a particular address would still be exposed. In addition to its security weaknesses, direct encryption incurs significant delays and performance overheads due to the high AES latency (typically tens of cycles) encountered in the critical path of each access.

Counter-mode Encryption: unlike direct encryption, counter-mode encryption associates each memory block (e.g., 64 bytes) with an encryption counter that is used along with the block address to form *initialization vector (IV)*, as shown in Figure 2.2. On each encryption of a particular block, its associated counter is incremented. Instead of directly encrypting the plaintext using the AES engine, counter-mode encryption uses the IV to generate a spatially (due to the use of address) and temporally (due to the use of counter) unique encryption pad that will be merely XOR'ed with the plaintext to complete the encryption process, as shown in Figure 2.1-b. The decryption process is similar except that the ciphertext will be XOR'ed with an encryption pad. Note that the encryption pad can be pre-generated without waiting until the arrival of the ciphertext, which can effectively hide the decryption latency. For the rest of the dissertation, we will use counter-mode encryption due to its security and performance advantages of direct encryption.



(a) Direct Encryption (b) Counter-mode Encryption

Figure 2.1: Difference between direction encryption and counter-mode encryption.

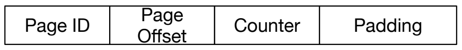


Figure 2.2: The fields of the initialization vector used in counter-mode encryption.

The encryption counters used to form the IVs correspond to memory blocks that are stored in memory. Thus, to complete an encryption/decryption operation, the encryption counter corresponding to the block to be read/written needs be fetched from memory to generate the encryption pad used to complete the decryption/encryption. Note that the encryption counters themselves are not confidential, since the encryption key is unknown to the adversaries. However, fetching encryption counters on each memory access can encounter significant performance overheads, and thus a counter cache is generally used to cache counter blocks. For storage efficiency, counters are typically packed in 64-byte blocks that contain 64 counters organized in a split mode, one 64-bit major counter and 64 7-bit minor counters. Each counter block covers the encryption counters of 64 cachelines, i.e., 4KB page when using 64-byte cachelines(56). While encryption counters themselves are not a secret, tampering with them or replaying them can comprise the system's security due to known-plaintext attacks. Thus, it is essential to protect the integrity of encryption counters to allow secure usage of counter-mode encryption.

2.1.2 Integrity Verification

Secure processors need not only to protect the confidentiality but also the integrity of the data when stored off-chip (i.e., outside the trust base). Also, as mentioned earlier, the encryption counters used to protect confidentiality need to have their integrity protected as well. Thus, modern secure processors implement an integrity verification mechanism to detect tamper or replay of data or encryption counters. However, typical authentication mechanisms such as associating each block with a message-authentication code (MAC) would fail to prevent replaying old content along with their MAC. Thus, integrity trees, typically called *Merkle Trees*, are used.

A Merkle Tree protects the integrity of the memory by a tree of hashes/MACs where the root is securely stored inside the processor chip. Thus, any memory update would lead to update the corresponding tree path and the root, as shown in Figure 2.3. Similarly, any memory read, once fetched to the processor, needs be verified by calculating its hash/MAC and subsequently verify with its parent hash/MAC (if verified), otherwise the parent needs to be verified through the grand-parent etc. up until reaching a verified node in the path. Note that once a tree node has been verified and inserted in the processor cache, it remains verified as it is no longer vulnerable to external attacks, until it gets evicted.

Generally, there are two types of integrity trees: ***Merkle Tree (MT)*** and ***Tree of Counters (ToC)***. As shown in Figure 2.3, MTs can be thought of as a tree of hashes where the protected parts are the leaves, and then levels of hashes are built on top of each other until all collapsing into a single value, the root. On the other hand, while ToC leaves are the protected data (or encryption counters), the ToC nodes consist of counters (also called versions) and a MAC that is calculated on them and their parent counter/version in the next level. Thus, the ToC's root is also a node that contains counters/versions that are used as input to the MAC stored along with the counters/versions in their immediate children nodes and so on. Figure 2.4 depicts a sample ToC. In the presence of a large number of parallel MAC units, ToC can update all levels in parallel, whereas MT updates propagate serially to the root. However, while ToC is used in Intel's SGX, its ability to leverage such parallel updates for large memory capacities (e.g., large enclaves) is restricted by the power and area overheads that accompany the use of tens of MAC engines per memory controller. Moreover, ToCs significantly complicate crash recoverability in persistent memory(65). While the solutions proposed in this work apply to both ToC and MT, we limit the scope of discussion for the rest of the dissertation to MT due to its simplicity, however, ToCs can leverage our approaches as is to improve the performance without the need for an impractical number of MAC engines required for parallel updates in large capacity memories.

After fetch and verification, MT nodes are typically cached inside the processor chip in a metadata cache, which allows speeding up the verification process of future accesses. Moreover, such caching allows faster updates to the MT in case of write operations. In general, there are two ways to update MT in the presence of metadata cache, *eager update and lazy update*. In a lazy update scheme, updates propagate upwardly on upon the eviction of a dirty node. For instance, a MT node in level 2 is updated with a new hash value only

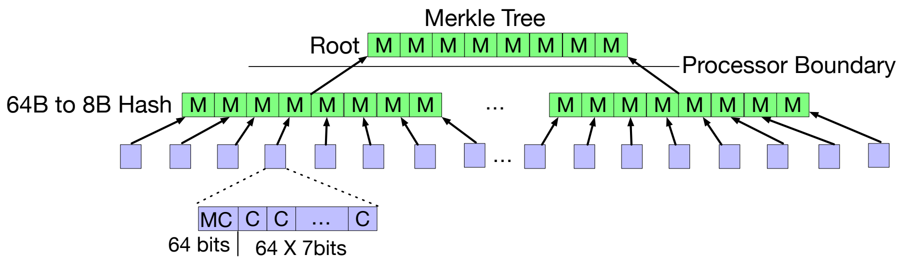


Figure 2.3: Merkle Tree

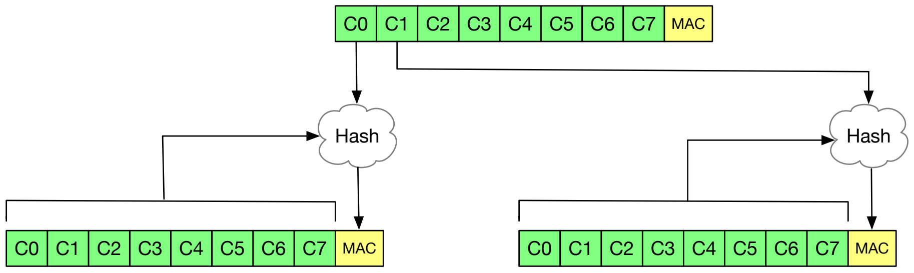

C0 C1 C2 C3 C4 C5 C6 C7 MAC
Hash
Hash
C0 C1 C2 C3 C4 C5 C6 C7 MAC
C0 C1 C2 C3 C4 C5 C6 C7 MAC

Figure 2.4: Tree of Counters

when one of its updated children in level 1 gets evicted from the metadata cache. In such a scheme, the root of the is not always up-to-date, however since the updates to MT nodes are eventually propagated to the root, and such updated MT nodes will be used in the verification, lazy update retains the same security guarantees of updating the root on each memory update. On the contrary, an eager update scheme updates the whole affected tree path up to and including the root on each memory update. While lazy update is clearly a better option than eager update for conventional memories, it introduces major crash consistency problems when used with persistent memories; the root is not up-to-date and the content of the metadata cache will be lost during a crash, thus secure recovery is infeasible(12; 65). Therefore, eager MT update is generally used with secure NVMs(65; 12; 24).

Since both encryption counters and data need to have their integrity protected, state-of-the-art solutions leverage a Bonsai Merkle Tree (BMT)(47), as shown in Figure 2.5, which allows integrity verification of data and encryption counters by merely using integrity tree over encryption counters while associating each data block with a MAC value that is calculated over the MT-verifiable counter, address and ciphertext. By doing so, the storage overheads of integrity protection is minimized compared to building two trees, one over data and the other over encryption counters.

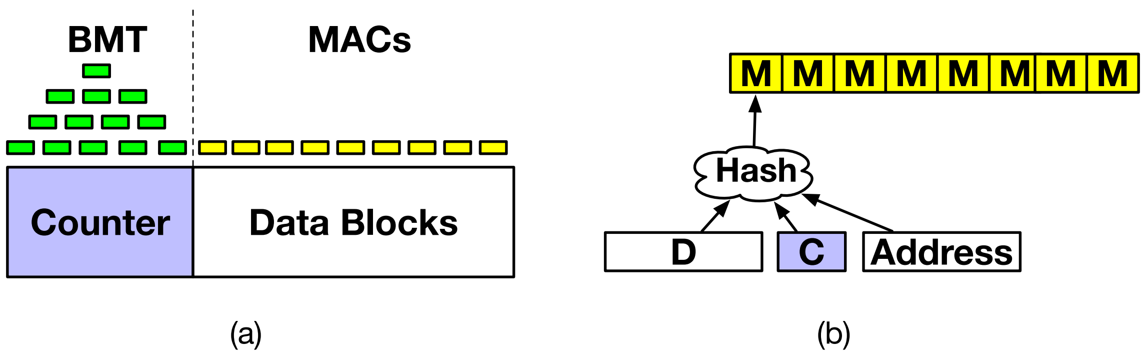
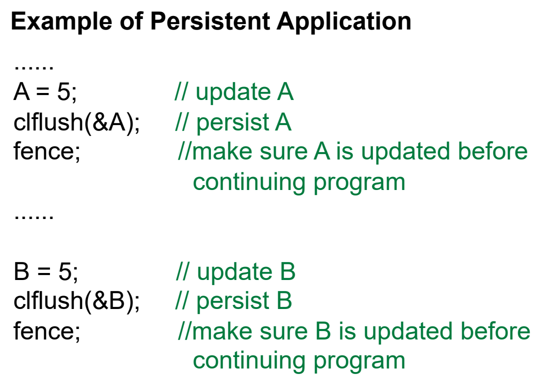


Figure 2.5: Bonsai Merkle Tree

2.2 Secure Persistent Memory

2.2.1 Persistent Memory

Persistent Memory generally refers to memory that can retain its content after losing power. Persistent memory can be realized in different ways varying from solutions as expensive as sufficient battery to flush the whole DRAM content to a flash storage, e.g., in NVDIMM-N, to solutions that leverage the high-capacity and persistence features of emerging non-volatile memories (NVM, e.g., Intel's DCPMM and JEDEC's NVDIMM-P standard). The latter approach has generally received more interest due to its lower access latency than storage-like memory, high-capacity exposed to the system, and the reduced costs and area overheads for battery or back-up power source (e.g., large capacitor). Such properties make it possible to use NVM as main memory to enable high performance and crash-recoverable applications, such as database workloads(44). Applications utilize the persistency feature of persistent memory is called persistent applications, which store objects in persistent memory and maintain a way to reconnect to stored persistent objects across application runs. However, just replacing the volatile main memory(e.g., DRAM) with persistent memory is not enough to guarantee data retention due to the discrepancy of memory hierarchy: volatile on-chip memory (i.e., cache) and non-volatile main memory. In modern processor, without cache bypassing or explicitly writing back or flushing cache blocks, data stays in the cache until natural evictions. Such evictions are transparent to applications. Therefore, upon a crash or power outage, cached data that has not been updated in persistent memory will be lost. To bridge this discrepancy between volatile cache and persistent main memory, persistent applications explicitly flush cached critical data to persistent memory. The basic idea is to allow applications to control the timing of writing back a critical cache block. As shown



Example of Persistent Application

```
......
A = 5;           // update A
clflush(&A);     // persist A
fence;           //make sure A is updated before
                   continuing program
......

B = 5;           // update B
clflush(&B);     // persist B
fence;           //make sure B is updated before
                   continuing program
```

Figure 2.6: Example of Persistent Application

in Figure2.6, after storing critical data, a cache maintenance instruction (*clwb/clflush*) is inserted to persist data to main memory. To guarantee ordering of critical persistent memory writes, memory fence is issued after cache maintenance instruction. Persistent applications use such long-latency cache maintenance instructions and memory fences to make sure that writes to persistent data reach the persistent memory, in a way that is crash consistent. Current persistence libraries, e.g., Intel's Persistent Memory Development Kit (PMDK)(33), provides users with such mechanism to explicitly persist updates.

In basic persistent memory architecture, persistence domain only includes NVM, as shown in Figure 2.7(a). Persisting data by flushing them all the way to persistence domain includes only NVM can significantly degrade performance due to the high write latency of NVM. Therefore, extra support is added to extend the persistence domain to include a small write buffer inside the processor chip, called write pending queue (WPQ), as shown in Figure 2.7(b). By relying on a backup battery or ultra-capacitors, the WPQ can be flushed to NVM when a power outage is detected. Since persisting data becomes as fast as flushing cachelines from caches to the WPQ, it eliminates the significant delays in the critical path that would have occurred due to persistent memory write latency. The support for additional reserved power to flush the internal WPQ buffer is called Asynchronous DRAM Refresh (ADR). To further improve the performance of persistent application and to improve programmability (i.e., eliminating memory flushes and fences), persistence domain can be further extended to include cache hierarchy, as shown in Figure 2.7(c). Such feature is referred as enhanced ADR (eADR). With larger battery than ADR, upon detection of crash or

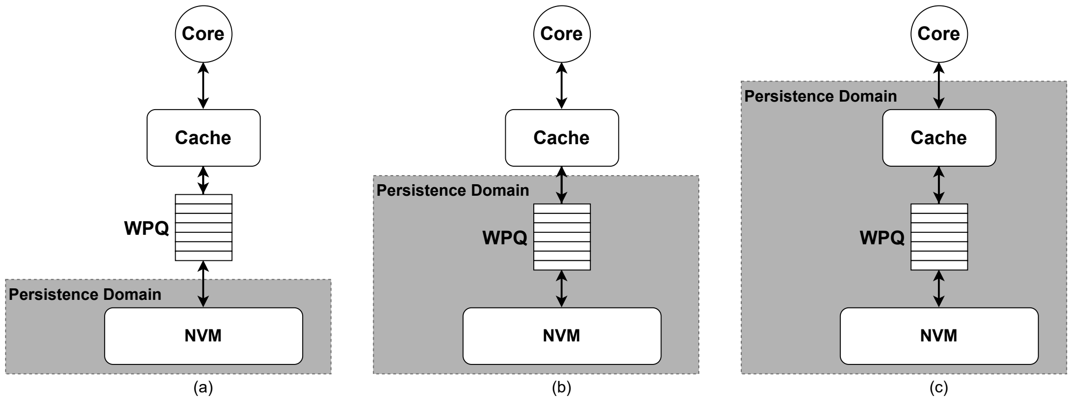


Figure 2.7: Persistent Memory Systems with Different Persistence Domains

power outage, entire cache hierarchy is persisted by flushing to persistent memory. We refer to systems that include such a feature (including cache hierarchy in persistence domain) by Extended Persistence Domain (EPD) memory systems.

2.2.2 Persistent Security

Securing On-Chip Persistent Domain

To reduce overhead of leveraging persistent memory for its feature of data persistency, persistent domain is extended to include on-chip WPQ with extra battery power. Such on-chip persistent content is flushed to off-chip persistent memory upon crashes therefore their content also needs to secured. One common assumption in all prior works is that crash consistency support needs to occur before considering the data persisted, i.e., the completion of a data persistence operation. Thus, a data persistence operation incurs expensive cryptographic operations, e.g., encryption and integrity tree updates, before the application can proceed to the next instruction. In persistent applications, Data persistency mandates flushing the data to be persisted all the way to the persistence domain (e.g., WPQ), hence the expensive secure operation is in the critical path of data persistence operation. Thus, it is important to minimize the overheads of data persistence operations in such workloads.

With even larger battery power, persistent domain can be further extended to include on-chip cache. In secure persistent memory with cache in persistent domain, cache content needs to be flushed to off-chip memory in a secured fashion upon crashes. In such systems, the insertion of data in the persistence domain is directly in the critical path as it corresponds to every write operation in the cache hierarchy (including L1 cache updates); the whole cache hierarchy is assumed persistent. Hence, the persistent security support is shifted towards ensuring that flushing the content of the cache hierarchy is done in a crash-consistent way with its corresponding security metadata. Oblivious to the amount of back-up power needed upon outage, one can safely assume that the cache hierarchy contents can be protected (i.e., encrypted and integrity-verifiable) and all security metadata updates can be flushed. However, as we will show in Chapter 4, the amount of extra work needed to enable the aforementioned solution can be prohibitive, especially with most data center vendors aiming to minimize their carbon footprints and the maintenance/deployment challenges for per-server batteries(23).

Handling Secure Metadata

To enable secure persistent memory, the data and its corresponding security metadata must be persisted atomically. Alternatively, there should be a mechanism to recover/infer such security metadata in a secure fashion. The BMT can be inferred, reproduced and verified through the root(12; 65). The BMT recovery process can be sped-up using low-overhead tracking as shown in prior work(65). However, the encryption counter and data MAC must be recovered and cannot be inferred from other elements. For instance, the MAC uses the counter however the MAC is also calculated over the data, and thus the most-recent MAC must be persistent along with the data to verify its integrity.

Prior works leverage ECC bits written with data to store MAC(49; 59) or (part of) encryption counter(60; 65; 12; 19; 58; 27). For instance, in Osiris(60), the encryption counter is embedded in the ECC bits through encrypting ECC along with the plaintext. Upon decryption, if a wrong/stale counter is used, the ECC will be unrelated (large number of code words will indicate error with high probability), and thus a stale counter can be identified. By limiting the divergence between the persisted counter in memory and the counter used for encryption, a limited number of trial is used until the counter used for encryption is recovered for each block. For MACs, Osiris assumes an independent MAC chip that can also be written concurrently with data. Most state-of-the-art solutions in secure NVM (60; 65; 12; 19; 58; 27; 66; 64; 59) builds on the idea of co-locating such security metadata with data to minimize the write amplification for implementing crash-consistent secure NVMs. In this paper, we address the problem of crash-consistent secure memory with a more generic and realistic emerging memory interfaces where no host-accessible ECC are available as in DDR-T(34; 1), CXL-based modules (e.g., SK Hynix 3DVXP(61)), and DDR5 modules in systems where the DDR5 on-die ECC support is considered sufficient(41), i.e., no host-managed ECC are used.

2.2.3 Threat Model for Persistent Memory Systems

Threat model in this dissertation is similar to the state-of-the-art work on secure persistent memory(65; 12; 60; 62; 13). Specifically, we assume a trusted processor, i.e., attackers cannot probe internal wires and caches within the processor chip. Moreover, similar to prior work, we only consider external attacks where (timing, power, electromagnetic) side-channel attacks and access control bypassing that leverage software and hardware bugs within the processor chips are out of scope. There exists a large body of work that addresses such internal attacks, and thus our work mainly focuses on external attacks. In our threat model, external attackers can snoop the memory bus, scan the memory module, tamper with the memory data and responses to the processor's requests. Moreover, while part of the tampering threat, attackers can attempt to swap memory locations' values. Specifically, the following attacks are considered in our threat model: spoofing attacks, relocation attacks and replay attacks. In our threat model, the memory content could be overwritten using some fake and arbitrary content, and can also be rolled back to an old version. Moreover, attackers can replace the content of one memory block with the content of a memory block in a different location. Thus, the works in this dissertation need to detect such attacks.

2.3 Memory Confidentiality in Embedded Systems

Internet-of-Things (IoT) devices are expected to proliferate in a wide range of sectors, varying from healthcare, appliances, military, and agricultural applications. Such widespread usage of IoT devices, and the nature of environments they are deployed in, raise major security and privacy concerns. In particular, IoT devices are expected to process personal information, from personal health-related data to sensitive military information. Moreover, as IoT devices are exposed to a larger attack surface during their lifespan, e.g., physical space access and maintenance, they can be a target for attackers to realize surveillance systems and compromise sensitive and personal data. Protecting the confidentiality of data is imperative for IoT devices.

2.3.1 Memory Systems in Embedded Systems

Memory resources in embedded systems can be divided into two parts: on-chip memory resources and off-chip memory resources. The on-chip memory resources include read-only memory (ROM) for instructions, random-access memory (RAM) and cache. Cache is a hardware-managed memory resources while RAM is manageable by software running on MCUs. The reason to have a software-managed on-chip RAM is that in embedded systems, sometimes it requires accurate and deterministic execution time in part of the application and using cache makes it difficult to predict the timing. Some MCUs provides faster on-chip RAM, whose access speed is comparable to cache, e.g., ARM tightly coupled memory (TCM)(10). On-chip RAM has disjointed memory address to the off-chip memory. Moreover, embedded systems commonly have embedded flash, within the processor chip, which allows permanent storage of certain data and codes, including initial firmware code.

Commonly, Memory Protection Unit (MPU) is used as the access control method in embedded systems; the physical address space is divided into regions, with each region can be programmed to have different access permissions (e.g., read-only, non-readable & non-writable) for different applications(7). When application violates the rules enforced by MPU, a fault signal is raised and MPU fault handler is invoked. Certain systems with more relaxed constraints can afford a full Memory Management Unit (MMU) that can capture access permissions at page-level and enable address translation, they are rare in IoT systems and embedded system world. For Chapter 6, we use a system with MPU as representative to embedded processors with constrained budget.

2.3.2 Threat Model for Embedded systems

Similar to other secure memory works(27; 66; 25; 65; 15; 40; 60; 4), this dissertation considers the Micro Controller Unit (MCU) as a Trusted Computing Block (TCB). Therefore, the internal FLASH memory and RAM that are part of the MCU are considered trustworthy, whereas external, off-chip memory is untrustworthy. In general, we assume attacks can be mounted against the external memory from the outside world. The work discussed in Chapter 6 prevents passive attacks, i.e., attacks that attempt to passively snoop memory bus or scan memory content via specialized tools or physical access to the system.

CHAPTER

3

DOLOS: IMPROVING THE PERFORMANCE OF PERSISTENT APPLICATIONS IN ADR-SUPPORTED SECURE MEMORY

To hide the expensive write latency of persistent memory, a small on-chip write buffer, Write Pending Queue (WPQ), is made persistent with battery power to flush its content to persistent memory upon crash / power outage. Such systems with battery-backed WPQ is reffered as ADR-supported systems. In secure persistent memory systems, since WPQ is leaked to off-chip untrustworthy persistent memory upon crash, its content needs to be encrypted and integrity-verifiable. One common assumption in all prior secure persistent memory works is that memory security operations and crash consistency support needs to occur before considering the data persisted. Such ordering of memory security operations and persisting data severely damages performance of persistent applications, which frequently use data persistence operations.

To solve the problem, we observe that it is possible to defer the security operations to occur towards the eviction from the WPQ, and thus incur minimal overheads in the critical path (insertion in the WPQ). However, the ***fundamental challenge*** is how to ensure that the security and recoverability guarantees of NVMs are met with minimal to no changes to the standard ADR power budget and circuitry supported in persistent memory platforms. Naive implementations that assume all security metadata operations can be done at the WPQ draining time (i.e., power failure detected) would fail to meet the ADR requirements due to the power-consuming cryptographic operations in addition to potentially tens of reads and writes of security metadata for each entry to be drained from the WPQ. Thus, our **goal** is to devise a novel secure NVM controller that leverages ADR capabilities to minimize the latency of data persistence operations, while also maintaining the security and recoverability requirements.

In this chapter, we propose ***Dolos***, a novel secure NVM controller that elegantly and securely shifts the overheads of secure operations to occur after eviction from the processor's internal persistence domain (i.e., WPQ buffer). By doing so, the majority of overheads due to security operations are removed from the critical path of data persistence operations, and thus exploit the otherwise untapped potential of ADR-backed WPQ. Dolos design is based on our observation that the protection of the contents of the WPQ can implemented in a two-step fashion, one step that is extremely lightweight at the time of inserting a write entry in the WPQ, which is in the critical path of the data persistence operations. While the second step is at the time of eviction from the WPQ, which essentially integrates the security metadata updates with the rest of the secure NVM's state. While the second step happens on each eviction from WPQ during the run-time, it can be skipped during WPQ draining due to ADR activation, i.e., detection of power outage. In other words, we provide two separate execution paths for evictions from WPQ, one during normal run-time and one during WPQ draining stage. The run-time path involves expensive operations and security metadata updates, whereas the WPQ draining path involves almost no extra operations beyond writing the WPQ contents to the NVM, i.e., complying with the standard ADR support for flushing WPQ. By doing so, Dolos adds minimal latency (the first step) to writes on their way to the persistence domain (i.e., completion of data persistence operation). Dolos leverages several novel mechanisms that cleverly hide the overheads for protecting the contents of the WPQ such that they can be merely flushed to NVM in case of a power outage while ensuring security, crash recoverability, and ultra-low latency in the critical path of persistent workloads. Dolos can be orthogonally integrated with prior secure NVM works that optimize memory backend operations (e.g.,(12; 65; 40; 4)), where Dolos minimizes the the latency of the memory front-end operations (insertion into the WPQ), and hence significantly improves persistent workloads' performance.

3.1 Motivation

To motivate the design of Dolos, we now consider the impact of a secure processor architecture with NVM and its performance implications on persistent applications. Figure 3.1 shows several possible secure processor architectures with NVM. The most basic is shown in Figure 3.1(a). The secure processor is inside the TCB where as off-chip NVM is not trusted, and the persistency domain is solely off-chip NVM. Writes have to go though a security unit before leaving processor.

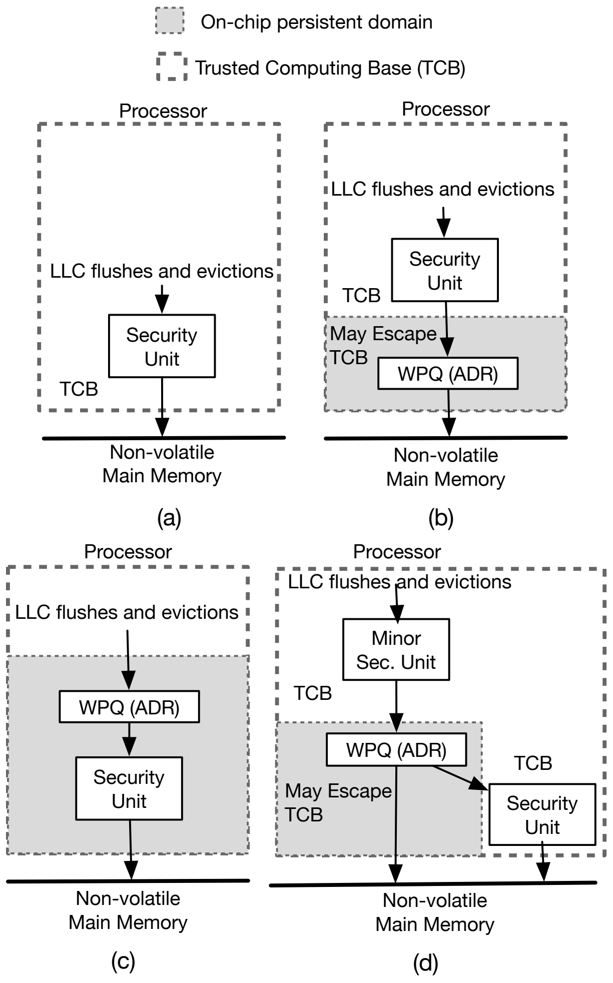


Figure 3.1: Models of Secure Memory and Secure Persistent Memory: a) Secure Memory Model without ADR, b) Secure Persistent Memory Model that cannot take full advantage of ADR, c) an infeasible approach that requires the persistency domain to subsume the Security Unit, and d) our approach: lower latency Minor Security Unit protects WPQ entries, major Security Unit protects memory.

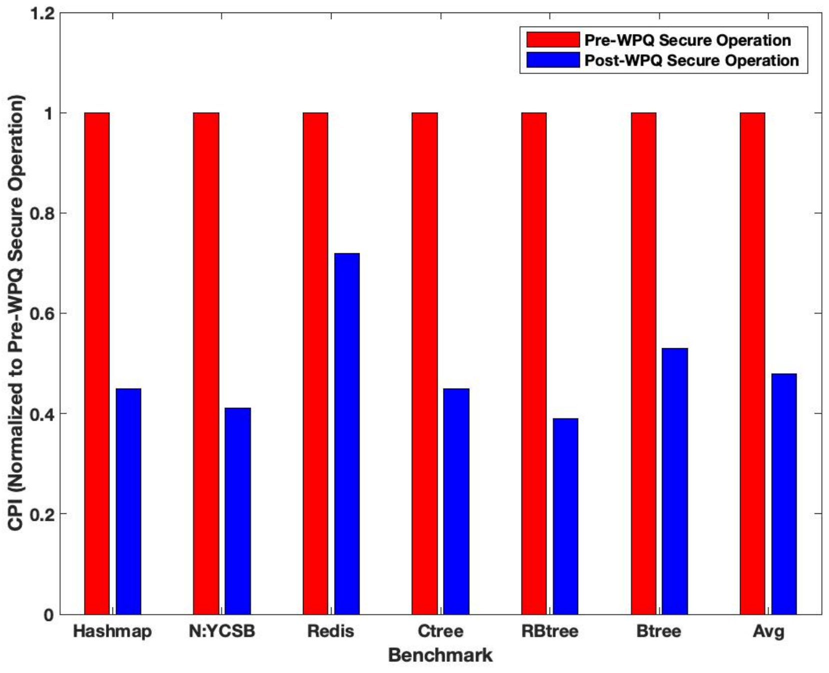


Figure 3.2: CPI between placing security process before and after WPQ.

The introduction of ADR, bringing the WPQ into an on-chip persistency domain, leads to the the two architectures in Figure 3.1-b and Figure 3.1-c. In these architectures, the persistency domain is the whole off-chip memory and the ADR-enabled WPQ. In Figure 3.1-b, the security unit is placed before the WPQ, thus it protects the whole persistency domain, even the WPQ. Upon a crash, content in the WPQ is immediately flushed outside the TCB to off-chip memory since it must have been previously protected by the security unit. This design imposes large overheads on persist operations because they must first pass through the security unit before entering the WPQ. The flushes and fences used to enforce persistent memory models must wait for flushes or writes to reach the WPQ to complete, and the security unit's latency will be added to that delay, slowing down persist operations. This adds up to considerable overhead.

Ideally, we would like to avoid these overheads by placing the security unit after the WPQ. Figure 3.1-c shows an architecture to hide the latency of the security unit by placing it after WPQ, thus persisting writes first and securing them later. Upon a crash, the WPQ content must go through the security unit before being written to memory. This implies that the ADR power supply is sufficient to complete the security operations of all pending writes in the WPQ.

To better understand the performance implications of performing security operations before persisting data, Figure 3.2 shows a performance comparison when performing the security operation before the WPQ (Fig 3.1-b) to a hypothetical scheme that allows delaying a security operation until after eviction from the WPQ (Fig 3.1-c). We use the same setup as described in Section 3.3 to collect these results. On average, we observe a 2.1x slowdown when inserting security operations' overhead (eager update of integrity tree and encryption) and fetching their security metadata before insertion to the WPQ. We conclude from this analysis that it is far better for performance to persist data as soon as it arrives at the memory controller, i.e., inserting it immediately in the WPQ. However, it is likely infeasible to reserve enough power to complete the security operations, all of their potential metadata updates in memory, and possibly fetch security metadata from memory, when a power failure is detected. The standard ADR support is limited to flushing tens of entries in the WPQ, and thus shifting security operations to occur while powered by ADR would likely fail to complete all operations on time. Meanwhile, extending ADR capabilities to have more power would require larger batteries, higher costs, and restrict the solution from working on systems with standard ADR support.

An ideal solution would allow writes to persist immediately into the WPQ without waiting to achieve high performance while maintaining the same ADR power budget as non-secure systems. Our design approximates the ideal by introducing a lightweight security unit to protect the WPQ, as shown in Figure 3.1-d, that reduces the delay added to writes. Now, the WPQ is protected by the Minor Security Unit which has a much lower latency, and this allows data to quickly persist. Upon a crash or power failure, the WPQ content can be immediately flushed outside the TCB because it has been encrypted, thus incurring no extra overhead on the ADR power budget. Entries in the WPQ are then decrypted and re-encrypted by the major Security Unit off the critical path of persistence. Overall, this design leverages the ADR-enabled WPQ for faster commit of secure writes while also complying with ADR power constraints.

3.2 Implementation

In the section, we will discuss the detailed implementation of Dolos. Dolos adopts two different security units (Minor Security Unit & Major Security Unit) to protect ADR-supported WPQ and persistent memory respectively. We will discuss the detailed design of these two units in the following sections.

3.2.1 High-level Overview of Dolos

Considering both the threat model and persistence scheme, any proposed design needs to achieve the following:

- **Ultra-low latency for persisting data:** we should minimize the time between the arrival of a write request to the secure memory controller and the time such a request is considered persisted.
- **Security during run-time and across crashes:** any data that reaches the persistence domain is expected be sent off-chip before or upon detection of a crash, and thus it needs to have its integrity and confidentiality protected.
- **Crash Consistency:** any data arrives to the persistence domain must be recoverable. Thus, the encrypted data along with any security metadata that are required to decrypt it and verify its integrity must be retained after a crash.

Thus, to meet these design requirements, we leverage a split security implementation, as shown in Figure 3.3. In this design, we ensure ultra-low latency for data persistence operations by adding a small security unit, *Minor Security Unit (Mi-SU)*, that is responsible of protecting the integrity and confidentiality of the WPQ content through novel optimizations that leverage the unique features of the WPQ: (1) its small size (2) the fact that its encryption pads can be pre-calculated at boot-up time; WPQ content encrypted by Mi-SU will be written to NVM only if a crash is detected. Moreover, Mi-SU meets the security requirements of its content upon crashes by encryption and integrity verification, however, it relegates the run-time protection to the *major security unit (Ma-SU)*. Finally, for crash consistency, Mi-SU ensures the recoverability of its encrypted and integrity-verifiable content in case a power failure event occurs. During run-time, before any entry is evicted from WPQ by Ma-SU, Ma-SU ensures that the entry has been persisted to NVM in a crash consistent manner by employing schemes like Triad-NVM(12) or Anubis(65). In this work, Ma-SU uses Anubis as the Ma-SU mechanism for crash consistency of run-time updates.

3.2.2 Minor Security Unit (Mi-SU)

Mi-SU is perhaps the most critical design element in Dolos due to its direct impact on the latency of data persistence operations. Thus, we will first discuss the possible design

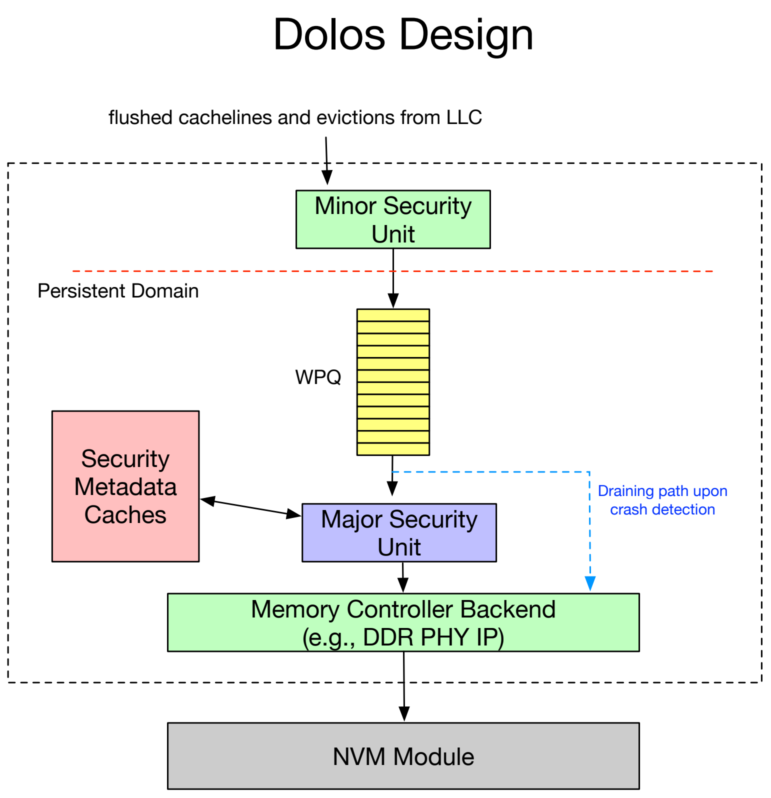

Dolos Design
flushed cachelines and evictions from LLC
Minor Security Unit
Persistent Domain
WPQ
Security Metadata Caches
Major Security Unit
Draining path upon crash detection
Memory Controller Backend (e.g., DDR PHY IP)
NVM Module

Figure 3.3: System Overview

options. In particular, we will discuss three design options that have different trade-offs between the effective WPQ size that can be used to buffer write requests and the amount of work needed before the write request is considered persisted. A discussion of each design option follows.

Design Option 1: One possible design is to use direct encryption before inserting an entry in the WPQ and calculating a MAC value on all the WPQ entries, similar to Merkle Tree. The encryption key used to encrypt the content of the WPQ will change upon boot-up (after recovering the previously flushed WPQ content). Such a scheme is sub-optimal due to the common issues of direction encryption (latency and security and MAC calculation latency to update the WPQ root); a 64-entry WPQ would require two MAC computations to update the tree root (assuming 8-byte MAC computed over each WPQ entry). This totals up to one encryption and two MAC calculations before insertion, which totals to 360 cycles (40 for encryption and 320 for two MAC calculations) in the critical path. An alternative option would be to use counter-mode encryption where each entry is associated with a unique counter value generated at boot-up time, and thus the encryption pads can be pre-generated. After encryption, the WPQ's Merkle Tree can be updated through two MAC calculations. However, even though this scheme replaces the encryption latency by a simple XOR operation with a pre-generated pad, this scheme still incurs two MAC calculations before insertion. Moreover, to be able to recover the content of WPQ after system restoration, the counters need to be recovered. Finally, guaranteeing no reuse of counters used to pre-generate pads highly depends on the perfectness of the random number generator used at boot-up time to generate such counters corresponding to each WPQ entry. Thus, we opt for using a persistent counter register that is incremented by the number of entries in WPQ at each reboot. By doing this, we can know the counters used for encryption of the previously flushed WPQ and guarantee the uniqueness of the counters that will be used for encrypting WPQ the next time it gets drained; each WPQ entry will be encrypted with the persistent counter register value plus the entry number. Note that even though the pre-generated pads will be used many times to encrypt the same entry in WPQ, it will be visible to the attacker only once, upon the draining event, and it will never be re-used again. Since this scheme leverages the ADR support to drain the whole WPQ, without the need to drain additional data, e.g., MACs of data, we call this scheme ***Full-WPQ-MiSU***.

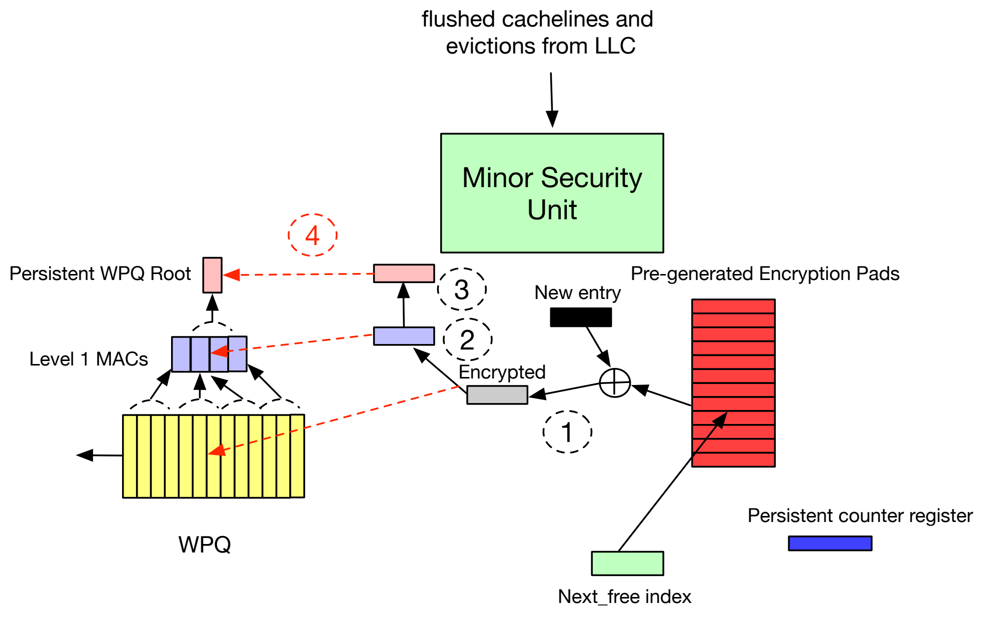

flushed cachelines and evictions from LLC
Minor Security Unit
4
Persistent WPQ Root
3
New entry
Pre-generated Encryption Pads
2
Level 1 MACs
Encrypted
1
WPQ
Persistent counter register
Next_free index

Figure 3.4: The Full-WPQ-MiSU scheme for Mi-SU.

As shown in Figure 3.4, processing a request arrival to Full-WPQ-MiSU design consists of the following steps: ① the pre-generated encryption pad of the free spot in the WPQ is

XOR'ed with the new entry. Note that this step is cryptographically secure since the pads are generated using AES CTR mode encryption where the counters are never repeated (each counter value will be used for a single draining operation), i.e., its generated ciphertext appears externally only once. Step ② involves using the ciphertext (along with its siblings) to re-calculate their parent L1 MAC. Later, in Step ③, the root will be re-calculated based on the value of L1 MAC. Finally, in Step ④, the new encrypted entry, the L1 MAC and the new root will be atomically written to the persistent WPQ Root, Level 1 MAC register, and the WPQ. Finally, since we manage WPQ as a circular buffer, the Next_time index will be incremented. Note that a *cleared* bit will be used along with each entry to indicate if it was fetched by and fully processed by the Ma-SU. The obvious overhead in this design is the need for two MAC calculations (steps ② and ③) in the critical path.

Design Option 2: One observation we make is that since the encryption counters of WPQ do not change when new WPQ-entries are inserted during the same run, using a scheme similar to a Bonsai Merkle Tree (BMT) can possibly minimize the MAC calculations to a single one. In particular, BMT in this context calculates a MAC over a WPQ entry and the encryption counter. However, since the persistent counter register can be used to securely deduce the counter used to encrypt each WPQ entry before the crash, we do not need to calculate a tree over the counters since we can securely recover their values. We can use such securely recovered counter values to verify the MAC values written with each WPQ-entry at draining time. Thus, the only way to forge a WPQ entry is to replay the internal persistent register, which is impossible since it is inside the processor. Accordingly, at recovery time, each WPQ is verified by calculating the MAC over the ciphertext and the internally-recovered corresponding counter. If the MAC value matches what has been written with the WPQ entry to memory, then the entry is successfully recovered. Note that when an entry is marked cleared by the Ma-SU, its MAC does not need to be re-calculated since re-writing it again upon recovery will not cause any security concern; the same ciphertext will appear. However, this scheme requires book-keeping the MACs that will be written with each WPQ entry upon draining, and thus would either require extra ADR support or limit the number of WPQ entries can be flushed upon crash. Since we are limited by the standard ADR, we opt for using a slightly smaller WPQ; since MACs are only $\frac{1}{9}$th of WPQ(8-byte for each 72-byte WPQ entry), we make only $\frac{8}{9}$ of the WPQ used for entries while the rest is for MACs. Accordingly, we call this design ***Partial-WPQ-MiSU***, where the trade-off here is smaller usable WPQ but only one MAC calculation (instead of two in Full-WPQ-MiSU).

As shown in Figure 3.5, processing a request arrival to Partial-WPQ-MiSU design consists of the following steps: ① Same as Full-WPQ-MiSU, the pre-generated encryption pad of the

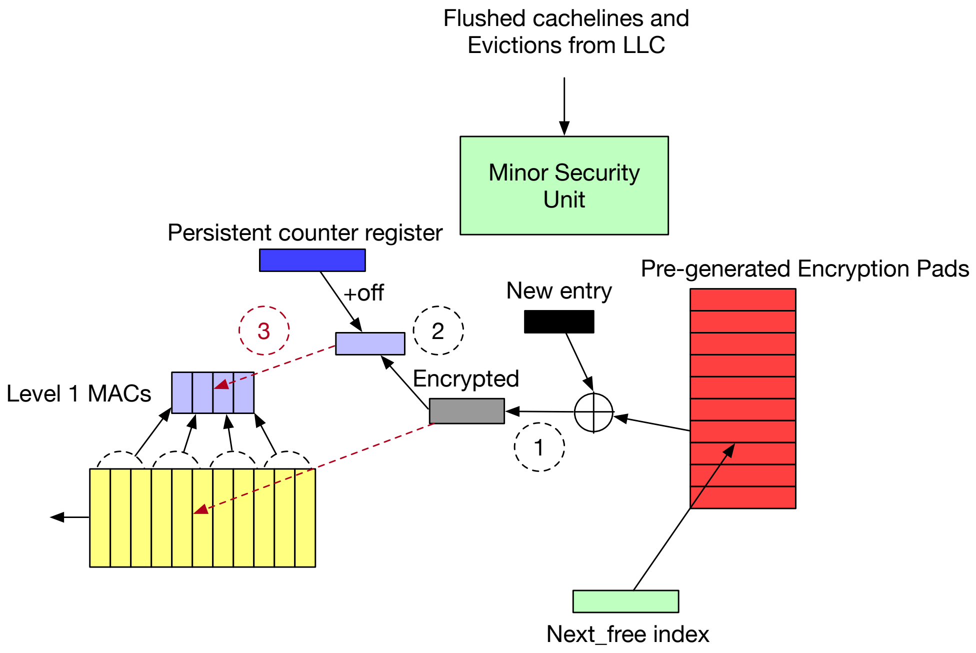

Flushed cachelines and Evictions from LLC
Minor Security Unit
Persistent counter register
+off
Pre-generated Encryption Pads
New entry
3
2
Encrypted
Level 1 MACs
1
Next_free index

Figure 3.5: The Partial-WPQ-MiSU scheme for Mi-SU.

free spot in the WPQ is XOR'ed with the new entry. ② Calculate MAC using the generated ciphertext and its own counter value. ③ Atomically update L1 MAC and encrypted entry. The obvious overhead in this design is the need for one MAC calculation (steps ②).

Design Option 3: To further reduce the secure latency of Partial-WPQ-MiSU before committing writes, the secure operation of the Partial-WPQ-MiSU can be delayed after committing a write by leveraging ADR to finish the remaining operation on the already committed write upon a crash. To avoid adding extra ADR budget, we reduce the number of WPQ entries to make up for ADR support for one secure operation in Partial-WPQ-MiSU, which is a single MAC computation. To maintain a reasonably-sized WPQ, we only allow one committed write with a delayed secure operation. We choose to allow a single delayed MAC operation based on our observation that the average WPQ request arrival time is 473 cycles (excluding idle time). We call this design ***Post-WPQ-MiSU***. Inside, the implementation of the Post-WPQ-MiSU is similar to Partial-WPQ-MiSU except that the timing of the secure operation (MAC calculation and XOR with encryption pad) for the new write to be allocated in WPQ. Post-WPQ-MiSU secures the write immediately after it is committed while Partial-WPQ-MiSU secures the write before it is committed. Note that even in the case of a power outage between the time the entry is inserted and its secure operation is completed, we reserve enough ADR to complete that secure operation (by using a smaller number of

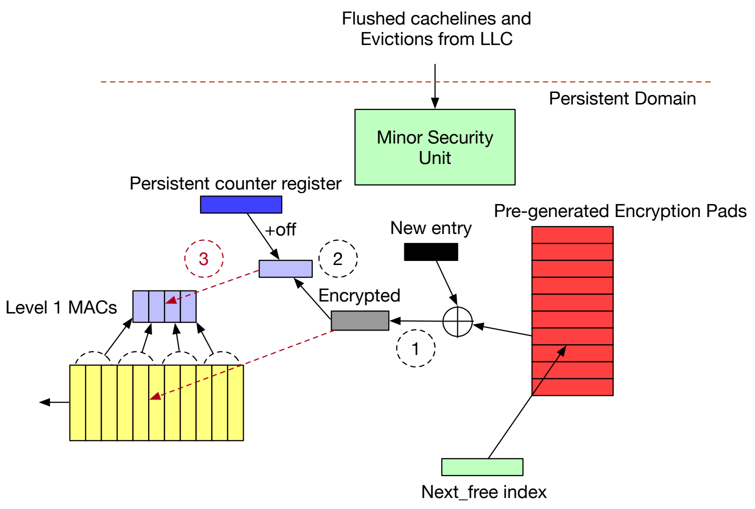

Flushed cachelines and Evictions from LLC
Persistent Domain
Minor Security Unit
Persistent counter register
+off
3
2
New entry
Pre-generated Encryption Pads
Level 1 MACs
Encrypted
1
Next_free index

Figure 3.6: The POST-WPQ-MiSU scheme for Mi-SU.

WPQ entries), and hence it is as secure as all other schemes. In Post-WPQ-MiSU, persistent domain starts from Post-WPQ-MiSU once a write request is accepted (i.e., MiSU is not full or busy), as shown in Figure 3.6. The trade-off here is almost zero overhead of secure operations at insertion time to persistence domain, but even smaller effective WPQ size (due to reserving some ADR for delayed secure operations).

Recovery scheme: Next, we discuss our recovery scheme for Mi-SU and WPQ. During boot-up, the processor fetches the flushed data of Mi-SU from NVM. In the case of the Full-WPQ-MiSU, it only fetches the WPQ content and verifies its integrity using the kept tree root. In the case of Partial-WPQ-MiSU and Post-WPQ-MiSU, it fetches MACs along with WPQ content and verifies its integrity by recalculating the MACs using the kept counter. As has been discussed, upon recovery from a crash, encryption pads will be updated with new values. Before that, the old WPQ content needs to be decrypted using old encryption metadata. Encryption pads are re-generated using the old counters. To avoid extra storage for the plaintext of WPQ content, WPQ content is drained to Ma-SU as it is decrypted. After all WPQ entries go to Ma-SU, counter and secrete key are updated and pre-generated pads are calculated.

3.2.3 Major Security Unit (Ma-SU)

In Dolos, Ma-SU performs the job of conventional security unit, responsible for full-memory protection before extracting an entry from the WPQ. Different from a conventional secure unit, it performs an extra XOR operation to decrypt WPQ content using the same encryption pad that was used upon insertion to the Mi-SU. Then, Ma-SU works on protecting data confidentiality and integrity, just as conventional security unit. In Dolos, there is no specific restriction on how to encrypt data and protect integrity, as long as the requirement for security and recoverability is met. For simplicity, the following discussion is based on the implementation of a counter-mode-scheme and BMT. To be able to recover, we implement the recovery scheme proposed by Osiris(60) and Anubis(65), which will be described in Section 3.4. In other words, before removing an entry from WPQ, its corresponding security metadata and any extra status information needed for recovery are atomically persisted as done in Anubis(65). Thus, at any point of time, a write request will be either recoverable through Mi-SU or persistent in a crash consistent manner through Ma-SU.

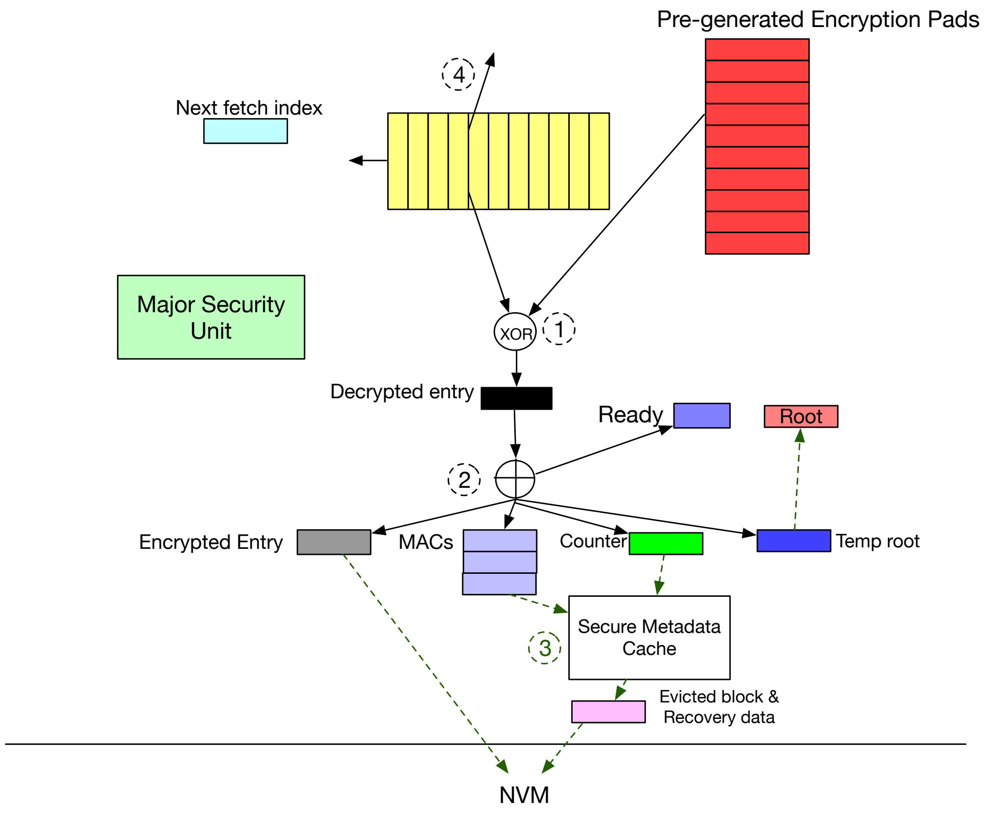


Figure 3.7: Ma-SU Scheme.

The steps of Ma-SU upon a single WPQ entry are shown in Figure 3.7. (1) Use the next_-fetch_index to fetch one WPQ entry, XOR its content with stored encryption pad to obtain the decrypted WPQ entry. Step ② involves encrypting data, calculating MAC and updating BMT nodes. Before overwriting the secure metadata cache and NVM, generated results (encrypted data, MACs, counter, temp root) are stored in persistent registers used as redo logging buffer (only set as ready when all needed updates are logged, e.g., ciphertext, tree root and intermediate nodes etc.). Once all tentative updates are logged (by the end of Step ②), Step ③ updates metadata cache and memory with tree/counter updates and recovery info (shadow tracker in Anubis(65)) along with the ciphertext, respectively. ④ Atomically set state and evict WPQ entry (advance next fetch index). Before working on the next WPQ entry, the ready bit of the redo logging buffer is cleared. Note that Steps ③ and ④ can be parallelized since once the redo logging buffers are filled we can redo the write request securely and in a crash consistent fashion, thus the WPQ entry managed by Mi-SU can be simply discarded and considered finished. Note that ③ and ④ do not need to occur atomically; the worst case is that the WPQ entry is not cleared while step ③ has written its corresponding ciphertext to memory. In that case, the same entry will be written again upon recovery but encrypted using a different pad than the one used by Ma-SU, and thus still secure but incurs extra work upon recovery. Note that the other scenario is that the WPQ entry is cleared but step ③, however this is straightforward as the updates are saved in the intermediate redo logging buffer, and thus can be performed again upon recovery.

Integrity tree type and update scheme: Both the eager update scheme and the lazy update scheme can be adopted in MaSU depending on the integrity tree type used to protect the main memory (Merkle Tree vs. ToC). As shown in prior work, for regular MT, it is sufficient to maintain an up-to-date root to enable recovery after crashes, thus merely updating the root eagerly and persistently while updating other levels only upon eviction is sufficient, as long as the counters are recoverable. Similar to prior works that use MT (AGIT(65) and Triad-NVM(12)) we assume the counters are recoverable using Osiris(60). Hence, we adopt AGIT(65) for MT where the root is eagerly updated. For ToC, as shown by prior works(4; 65), eagerly updating the root persistently is insufficient to recover the tree due to inter-level dependencies in SGX-style trees(65). Therefore, state-of-the-art schemes(4; 65) use an additional integrity tree (shadow tree) that is eagerly-updated to protect the cache of a lazily-updated ToC. Leveraging the parallelism of ToC to update all levels in parallel is left untapped due to the need to update all levels persistently to enable recovery, and thus a lazily-updated ToC covering memory with a MT-based integrity tree covering the cache is adopted for ToC-based integrity protection, i.e., we use Phoenix(4) for ToC.

Recovery scheme: In this paragraph, we will discuss the recovery scheme for Ma-SU.

During boot-up, the processor will check the ready bit. If the ready bit is not set, redo logging in the persistent buffer is discarded. Secure metadata is recovered to a state consistent with the value in the root register. Ma-SU resumes from step ①. If ready bit is set, secure metadata is recovered to a state consistent with value in temp root register. Ma-SU resumes from step ③. In this case, we assume that the corresponding WPQ entry is already evicted, in other words, step ④ is skipped during recovery so that an untouched WPQ entry will not be evicted by mistake.

3.2.4 Write Coalescing and Reads from WPQ

The encryption of WPQ entries (data and address) by the Mi-SU prevents look-up operations needed for maintaining consistency and optimizations like write-coalescing. To enable such operations, a volatile structure that maintains the address of each WPQ entry is added to enable quick look-ups of potential duplicates or to serve read requests (but note that another decryption would be needed). Another possible approach is to simply not encrypt the address part of WPQ entries, which would achieve the same level of security since an attacker can observe the addresses sooner or later regardless of whether or not a crash occurs. In either implementation, a read request that hits in the WPQ needs to be decrypted. Since such a decryption would merely take an XOR operation (one cycle) and because the chance of a hit in the small WPQ is minute, the additional overheads are negligible. Thus, in Dolos, we use a parallel volatile tag array for WPQ entries to enable write coalescing and for resolving reads to entries in the WPQ.

3.2.5 Security Discussion

During normal execution, the whole memory is protected by a conventional security unit, therefore the behavior of detecting such attacks is maintained. When there is a crash, the WPQ content protected by a dedicated security unit in Dolos is flushed into the memory. This dedicated security unit ensures detecting the attacks on WPQ content by assigning a unique counter value for each WPQ entry and calculating a MAC value over a new allocation in the WPQ entry with its associated counter. Note that the counters used to encrypt WPQ contents upon a crash are kept persistently inside the processor, and thus cannot be tampered with. Therefore, any unexpected change on the WPQ content will be detected.

3.3 Evaluation

In this section, we describe our evaluation methodology followed by our experimental results and analysis of Dolos.

3.3.1 Simulation Setup

We use GEM5(17), a cycle-level simulator to evaluate the performance overheads of Dolos. As illustrated in Table 5.1, we simulate a single X86-64 Out-of-Order core with 16GB DDR based PCM[1]. We also use six database benchmarks from WHISPER(44). For each benchmark, we fast-forward to where the transactions start and simulate 50000 transactions. We simulate all the integrity protection and data encryption aspects in both of Mi-SU and Ma-SU. We model both MT with eager update (as in AGIT(65)) and ToC with lazy update (as in Phoenix(4)). For the update of ToC, AES-GCM and conservatively assume parallel AES-GCM engines with a design and latency based on prior work(57), i.e., the update different levels of ToC happen in parallel. For the update of the small BMT in lazy update scheme and regular BMT in eager update scheme, we use MAC latency of 160 cycles and all levels are updated serially, similar to prior work(40; 12; 65). A counter cache and MT cache are included in Ma-SU. In our model, a single MAC computation takes 160 cycles, similar to prior work(40; 12; 65). Full-WPQ-MiSU has two MAC computations and both Partial-WPQ-MiSU and Post-WPQ-MiSU have one MAC calculation. Ma-SU has ten MAC calculations for eager update scheme and four MAC calculations for lazy update scheme. Unless mentioned otherwise, we use the state-of-the-art secure NVM controller, Anubis(65) – AGIT scheme, as our baseline and representative of the pre-WPQ secure NVM implementations where all secure memory back-end operations occur before persistence. This is denoted Pre-WPQ-Secure in some figures. Note that Dolos can be combined with any secure NVM scheme which can be used to further improve the Ma-SU performance; as mentioned earlier, we use Anubis(65) as the Ma-SU's implementation in Dolos. For all applications, we use 1024B as the default transaction size and eager update scheme unless stated otherwise.

[1]Due to the lack of full support of atomic x86 instructions in Gem5, several applications failed in multi-threaded mode and thus we limited our evaluation to single-core. However, since our scheme improves the latency of persistence operations, we believe that our evaluation is sufficient and can directly apply to the multi-threaded version of the workloads.

Table 3.1: Simulation Configuration Parameters

Processor	
Core	1 Core, X86, OoO, 4GHz
L1 Cache	2 cycles, 32KB,2-Way
L2 Cache	20 Cycles, 512KB, 8-Way
LLC	32 Cycles, 8MB, 16-Way
DDR based PCM Memory	
Size	16 GB
Access Latency	read latency 150ns. write latency 500ns.
Secure Memory Parameters	
Counter Cache	128kB, 4-way, 64B Block
MT Cache	256kB, 8-way, 64B Block
AES Latency	40 Cycles
Hash Latency in Mi-SU	160 Cycles in Partial-WPQ-MiSU&Post-WPQ-MiSU 320 Cycles in Full-WPQ-MiSU
Hash Latency in Ma-SU	160×10 Cycles for Eager Update; 160×4 Cycles for Lazy Update
Integrity Tree	8-ary Merkle Tree; 8-ary TOC
Tree Update Policy	Eager Update(Merkle Tree); Lazy Update(TOC)

Table 3.2: Number of WPQ Insertion Re-try Events Per Kilo Write Requests (KWR)

Benchmark	**Number of WPQ Insertion Re-try-KWR**		
	Full-WPQ-MiSU	Partial-WPQ-MiSU	Post-WPQ-MiSU
Hashmap	182.32	293.00	359.30
Ctree	88.19	207.22	285.24
Btree	106.55	214.17	280.80
RBtree	120.00	209.89	261.22
NStore:YCSB	1.09	68.55	181.95
Redis	106.93	215.10	274.43

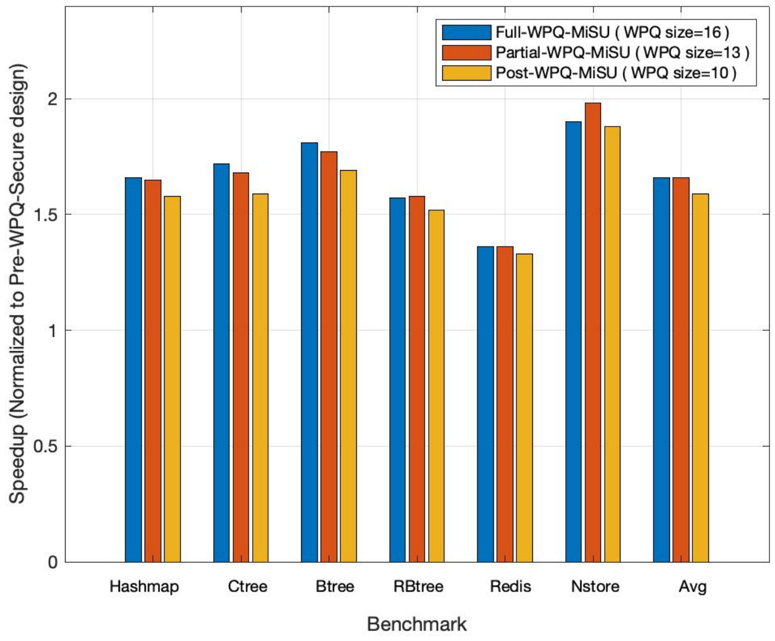


Figure 3.8: Speedup of Dolos with Full-WPQ-MiSU, Partial-WPQ-MiSU, Post-WPQ-MiSU using Eager Update Scheme(transaction size = 1024B)

3.3.2 Dolos Performance

Overall Performance in Eager Update Scheme

In this section, we show the performance improvement of Dolos when adopting three different designs of Mi-SU with Eager Update of Merkle Tree. Figure 3.8 shows an average speedup of 1.66×, 1.66×, 1.59× for Full-WPQ-MiSU, Partial-WPQ-MiSU and Post-WPQ-MiSU design. Since Partial-WPQ-MiSU and Post-WPQ-MiSU need ADR energy to also flush MACs and compute MAC, respectively, we use smaller number WPQ entries. In particular, the Full-WPQ-MiSU design has a 16-entry WPQ, whereas the Partial-WPQ-MiSU and the Post-WPQ-MiSU designs have 13-entry and 10-entry WPQ sizes, respectively. As shown in Figure 3.8, Post-WPQ-MiSU has a slightly less speedup than the other two designs. This is because of the high number of writebacks that arrived when the WPQ was full, mainly due to its smaller WPQ size. As shown in Table 3.2, the number of retry events (i.e. these occur when attempting to insert an entry in the WPQ when it is full) per kilo write requests (KWR) of the Post-WPQ-MiSU is much higher than the other two designs. For the case of Nstore, Partial-WPQ-MiSU has speedup of 1.98× and Full-WPQ-MiSU has speedup of 1.90×. This is because of smaller latency in Partial-WPQ-MiSU (1 MAC calculation vs. 2

MAC calculations) while the number of retry events to WPQ remains relatively low in both designs.

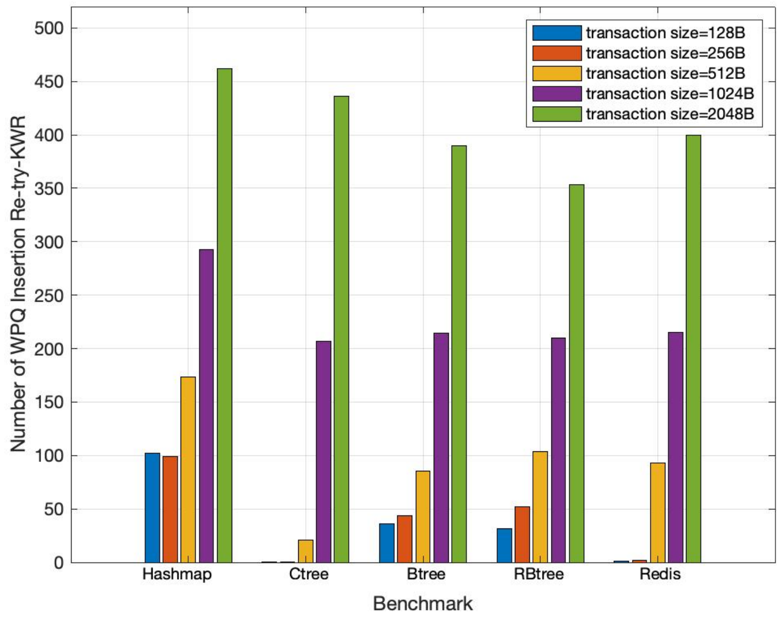


Figure 3.9: Number of WPQ insertion Re-try-KWR of Dolos with Partial-WPQ-MiSU on single transaction size of 128B, 256B, 512B, 1024B, 2048B

Variable Transaction Size

To study the performance improvement of Dolos when different transaction sizes are used, we run each application with transaction sizes of 128B, 256B, 512B, 1024B and 2048B. Compared to the baseline (16-entry WPQ in Pre-WPQ-Secure design), a 13-entry Partial-WPQ-MiSU design consistently achieves higher speed-ups in small transactions compared to large transactions. The reason is that large transactions can quickly fill the WPQ buffer and thus render the WPQ buffer less effective, as shown in Figure 3.9. However, we observe that even for transactions as large as 2048B, delaying secure operations to occur after insertion in WPQ can effectively improve performance, as shown in Figure 3.10. The main reason behinds this is that even for large transactions, large part of the transaction will be effectively buffered, whereas in the baseline each cacheline arriving to the memory

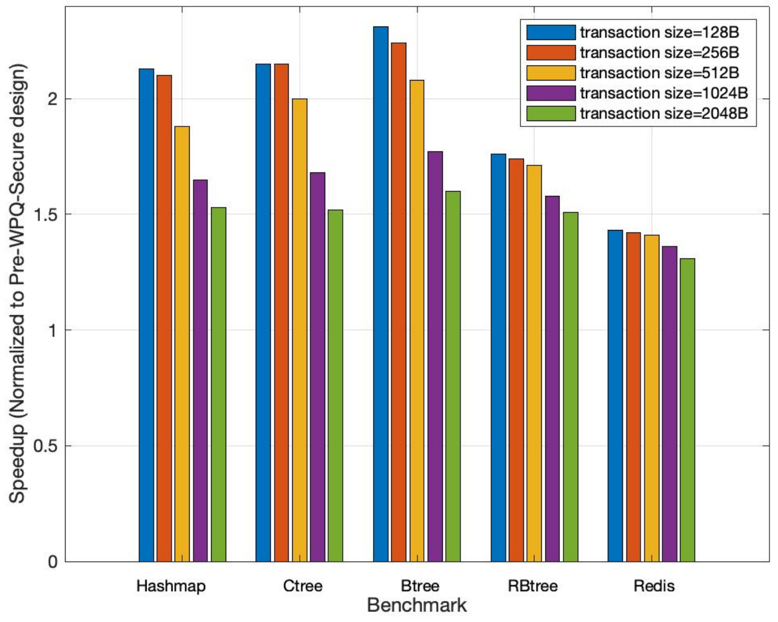


Figure 3.10: Speedup of Dolos with Partial-WPQ-MiSU on single transaction size of 128B, 256B, 512B, 1024B, 2048B

controller will need to go through secure operations immediately before insertion.

3.3.3 Sensitivity Study

Dolos' performance improvement is directly affected by the WPQ size; Dolos tries to effectively leverage the WPQ in hiding the data persistence latency. Thus, we change the WPQ size to better understand the robustness of Dolos in improving the performance. For a fair comparison, we use the full WPQ for the baseline (Pre-WPQ-Secure design) and a $\frac{8}{9}$ of full WPQ for Partial-WPQ-MiSU. As shown in Figure 3.11, the performance of Dolos with Partial-WPQ-MiSU improves when WPQ size increases. Dolos achieves an average speedup of 1.66×, 1.85×, 1.87× and 1.88× for WPQ size of 13, 28, 57 and 113. This is because the WPQ is occasionally full at 13 and rarely full at 28 or higher. Experimental results confirm that the average number of retry events per KWR is 201.32, 29.03, 13.55 and 11.08 for a WPQ of size 13, 28, 57 and 113. The speedup changes little with 28 or more entries in the WPQ.

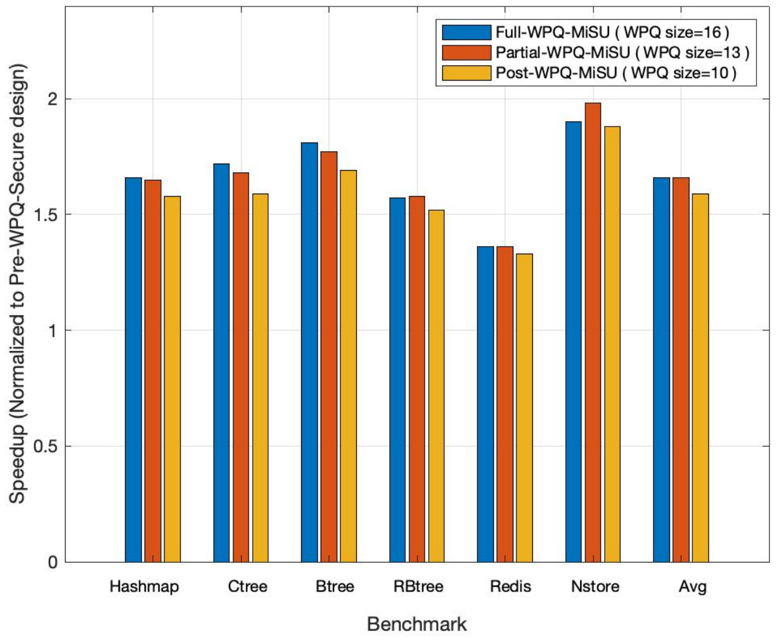


Figure 3.11: Speedup of Dolos with Partial-WPQ-MiSU on WPQ size of 13, 28, 57, 113 (transaction size = 1024B)

3.3.4 Dolos Performance in Lazy Update Scheme

In this section, to better illustrate Dolos performance, we show the performance improvement of Dolos when adopting three different designs of Mi-SU with Lazy Update of ToC(4). Figure 3.12 shows an average speedup of 1.044×, 1.079×, 1.071× for Full-WPQ-MiSU, Partial-WPQ-MiSU and Post-WPQ-MiSU design. In the lazy update scheme, Dolos with Full-WPQ-MiSU design has an obviously lower performance than the other two designs. This is because the lazy update scheme, MaSU has less MAC computation latency (computation of 4 levels Merkle tree over the secure metadata cache). Therefore, doubling the MAC computation latency in MiSU has an obvious impact on the performance. Simulation results show that Dolos with Post-WPQ-MiSU design has a slightly worse speedup than Dolos with Partial-WPQ-MiSU. This is because of the high number of writebacks that arrived when the WPQ was full, which is mainly due to its smaller WPQ size.

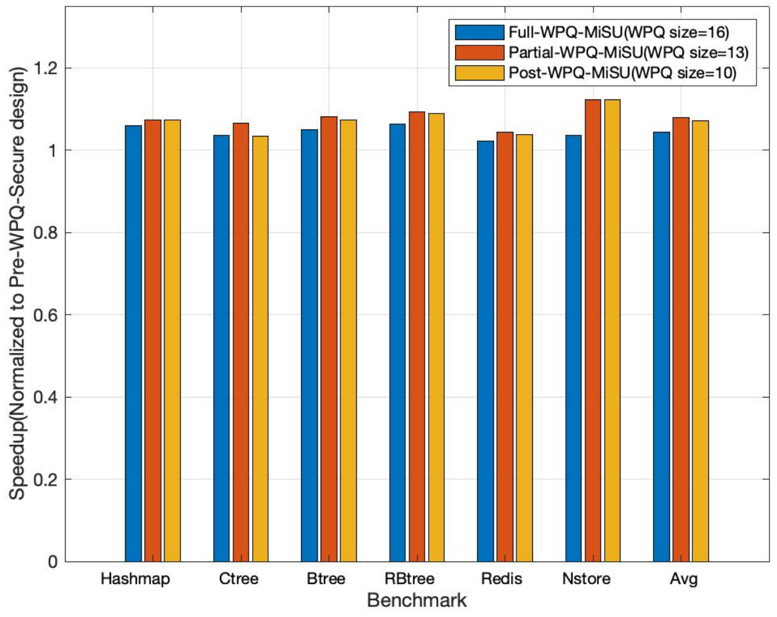


Figure 3.12: Speedup of Dolos with Full-WPQ-MiSU, Partial-WPQ-MiSU, Post-WPQ-MiSU using Lazy Update Scheme(transaction size = 1024B)

Table 3.3: Storage Overhead of Mi-SU

	Full-WPQ-MiSU	Partial-WPQ-MiSU	Post-WPQ-MiSU
Persistent Counter	8B	8B	8B
MACs	192B	128B	128B
Encryption PAD	72B * 16	80B * 13	80B * 10

3.3.5 Estimated Overheads of Mi-SU

We estimate the storage overhead and recovery overhead of Mi-SU using a 16-entry WPQ size. Table 3.3 lists the overheads for Dolos with Full-WPQ-MiSU, Partial-WPQ-MiSU and Post-WPQ-MiSU respectively. To allow the mechanism of write coalescing, additional volatile registers holding the unencrypted address information are needed. The additional area overhead is 8B * WPQ_SIZE. For the recovery time of Mi-SU, the main steps in recovering Mi-SU involves: 1. Read back WPQ content and secure metadata from NVM. 2. Regenerate encryption PADs using old secure metadata. 3. Decrypt and drain WPQ entries. 4. Update secure metadata and calculate encryption PADs. We assume a read latency for 64B block takes 600 cycles. In Full-WPQ-MiSU, only the WPQ content is read back, so the total read latency is 16*600 cycles. In Partial-WPQ-MiSU and Post-WPQ-MiSU, the WPQ content and two 64B MAC blocks are read back, however, there are fewer WPQ entries in these two designs. So the total read latency for Partial-WPQ-MiSU and Post-WPQ-MiSU is 15*600 cycles and 12*600 cycles, respectively. We assume that a single encryption pad generation takes 40 cycles. The latency for draining a single WPQ entry takes 2100 cycles(including NVM write latency and Ma-SU latency). Under this assumption, the recovery time for the Full-WPQ-MiSU takes 600 cycles * 16 + 40 cycles * 16 + 2100 cycles * 16 + 40 cycles * 16 = 44480 cycles, which is marginal (≈0.01ms).

3.4 Related Works

In this section, we will discuss the recovery schemes of secure NVM proposed in prior work. We will also discuss the relevant prior work on reducing the overheads of secure NVMs.

Secure Metadata recovery: In secure NVM, especially for persistent applications, data blocks need to be crash-consistent along with its associated secure metadata, i.e., counter and integrity tree. To reduce the extra memory traffic caused by secure operations, a secure metadata cache is implemented in the memory controller which introduces the crash consistency issue. Prior works(12; 65; 39; 60; 67; 24; 14) propose mechanisms to speed up recovery of security metadata. Osiris(60) utilizes ECC bits (co-located with ciphertexts) to verify the correctness of the counter value. By matching ECC re-calculated from decrypted data with the ECC stored with ciphertext, the processor can recognize the correct counters (we refer the readers to the original Osiris paper(60) for more details). Osiris has a long recovery time as it needs to rebuild the whole integrity tree based on recovered counters. Thus, Anubis(65) solves the problem by introducing a shadow cache in NVM to record the address information of the cached secure metadata. Upon a crash, the processor can pinpoint all potential inconsistent secure data blocks by utilizing address information kept in the shadow cache. While Dolos leverages Anubis for the Ma-SU implementation, it is orthogonal and can be integrated with any crash consistency scheme of secure NVMs. However, Dolos is fundamentally different in that it shortens the latency for inserting the data in the persistence domain than reducing recovery time or crash consistency overheads as in Anubis.

Reducing secure Non-volatile memory overhead: Prior works (48; 62; 67; 40) reduce the overhead of secure NVM in general. Morphable Counters(48) reduces the accesses of off-chip secure metadata by providing more counters per counter cacheline block. To reduce counter overflows, it creates two types of counter blocks depending on the number of zero-value minor counter in the block. DEUCE(62) propose a scheme to only re-encrypt changed words in a single cacheline, thus reducing flipped bits per writeback. Zuo et al.(67) leverage a lightweight hash function to quickly detect memory duplication on the granularity of a cacheline when dealing with writes. If a duplicate is detected, writeback and encryption is canceled by maintaining the mapping relationship between the canceled write and the duplicate cacheline. Janus(40) breaks down these backend memory operations including encryption, integrity protection and duplication detection and optimize them by executing independent operations in parallel and pre-execution. All of these works target reducing the performance overheads and writes to NVM, and how to efficiently implement secure memory backend operations to improve throughput (as in Janus(40)). Unlike prior works, Dolos aims to remove these overheads from the critical path of persistence operations by leveraging WPQ, while efficiently leverage any of the memory backend optimizations for the major security unit. In other words, Dolos can use any of the prior works however it adds a unique angle: *persist quickly then do the security operations in a quick fashion vs. do the security operations quickly then persist.*

3.5 Summary

In this chapter, we propose Dolos, a novel secure memory controller that allows delaying the expensive secure memory operations after the data is inserted in the persistence domain of ADR-supported persistent memories. Thus, it brings significant improvement in the performance of persistent applications. Based on our evaluation, Dolos improves the performance of persistent workloads by an average of 1.66× over the secure memory controller where the potential of WPQ backed by ADR support is untapped. To the best of our knowledge, Dolos is the first work that explores the design space and possible implementations for leveraging WPQ to improve persistent workloads in secure NVMs.

CHAPTER

4

HORUS: PERSISTENT SECURITY FOR EXTENDED PERSISTENCE-DOMAIN MEMORY SYSTEMS

As has been discussed in Chapter 1 and Chapter 3, to use persistent memory for data retention across multiple runs, applications need to use cache flushes and fences (i.e., persistent applications). To eliminate the need to call flushes / fences and enable DRAM-like crash consistency applications, entire cache hierarchy can be included in persistence domain with extra battery to flush cache content to persistent memory upon crash. Such systems is referred as Extended Persistence-Domain (EPD) memory systems. In secure persistent memory with EPD, cache content requires to be encrypted and integrity-protected when draining to the off-chip persistent memory. It is infeasible to secure cache content during runtime since it adds expensive security operation latency to every writes. Hence securing cache content is delayed until detection of crash.

However, securing cache content using conventional memory security solution during draining upon crash is challenging. Upon crash, locality of cache blocks is random and difficult to predict, hence in the worst case, it would incur significant increase in the number of memory accesses of security metadata required to secure cache blocks. Securing cache blocks may also cause a chain of security operations (i.e., updating all MT levels in eager update scheme and securing large number of evicted security metadata blocks in lazy update schemes). Finally, the increase in persistent memory capacity would also increase the number of levels, and hence increases the worst case number of operations required for draining cache hierarchy. Hence, ideally we need a solution that decouples the required backup power budget from the memory capacity, ensures high-performance memory security operations at run-time, incurring minimal extra power requirement compared to no security EPD systems, and ensures fast recovery of the system upon power connectivity.

Accordingly, the goals of this work are (1) define the backup power budget requirements for encrypted and integrity-verified EPD memory systems, (2) identify the trade-offs for back up power budget, run-time performance overheads, and recovery time (i.e., availability) for enabling crash-consistent and secure EPD systems, and (3) introduce novel mechanisms that reduce the overheads for ensuring recoverability while requiring minimal increase of the backup power budget of non-secure EPD systems. To define the backup power requirements, unlike traditional studies, we focus on the worst case scenario number of operations (e.g., memory writes) required upon the detection of a crash, and hence the amount of time the processor needs to be up upon a detection of crash. Hence, in our study we focus on optimizing the platform requirements to reduce the costs and area of encrypted and integrity-verified EPD memory systems, however without incurring run-time performance overheads or significant increase in recovery time.

In this Chapter, we propose a novel scheme, ***Horus***, which reduces the number of write and read requests of secure metadata and avoids updating the regular merkle tree during an EPD flush (i.e. the flush on crash of the EPD state). Horus adopts a split approach for data flushed during crash, different from the approach adopted during run time. It makes the security operation during the crash independent of the regular secure metadata, therefore, it avoids the memory requests of regular secure metadata and avoids updating the regular integrity tree.

4.1 Motivation

In this section, we demonstrate the challenges for implementing EPD systems with secure memory. As mentioned earlier, memory encryption and integrity verification are critical primitives in trusted execution environments. Such protections aim to protect the content of the memory during run-time and across crashes. In non-secure EPD memory systems, the cache hierarchy is flushed line by line to the memory, then the system is considered persisted and ready to go off. In case of inclusive last-level cache (LLC), the process can be simplified to flush all the dirty lines in LLC, since the coherence protocol will bring in any more recent version from upper-level caches. Hence, the EPD power hold-up budget should be designed to be sufficient to drain the maximum number of cache lines that can be dirty in the cache hierarchy, which can be as large as the number of cache lines in the LLC. Note that this can be significant when gigantic caches are used, e.g., AMD EPYC's 512MB V-Cache.

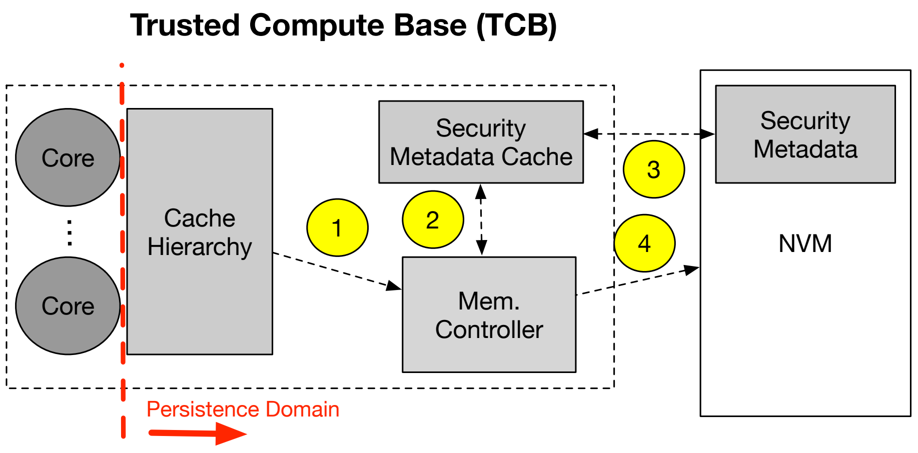


Figure 4.1: Draining the contents of cache hierarchy in secure EPD systems.

Similarly, in secure EPD memory systems, the contents of the cache hierarchy need to be flushed upon the detection of a power outage. However, each cache line needs to have its integrity (including freshness) and confidentiality protected all the time, including after power recovery. In its simplest form, draining the contents of a secure EPD memory system can be comprised of two steps: (1) draining the contents of the cache hierarchy while performing encryption, MAC calculation and integrity tree update on each cache line flush; i.e., treat LLC flushes similar to run-time main memory writes. Later, (2) flush/synchronize the security metadata cache updates, e.g., integrity tree and counter caches, to the persistent memory. Note that step (2) heavily depends on whether a lazy update or eager update schemes are used. For simplicity, let's assume that step (2) takes negligible number of operations compared to step (1). As shown in Figure 4.1, draining the cache hierarchy lines consists of the following steps for each eviction. (1) A cache line flushed from the cache hierarchy arrives to the memory controller. (2) Some time later, the memory controller needs to encrypt the line and update the integrity tree (whether lazily or eagerly) to ensure freshness protection, also updating the encryption counter and the MAC written with data regardless of whether such updates are done in the security metadata cache or also persisting to memory. Note that to complete this step, there could memory accesses required to fetch the security metadata (e.g., counter blocks, BMT path, and/or MAC block) to complete the protection. Even worse, fetching these metadata blocks to the metadata cache can lead to evicting other (possibly dirty) blocks from the metadata cache and cause additional memory writes. We refer to this as Step (3) in the figure. Finally, once the metadata needed to complete the protection of the cache line are fetched and updated (whether in the cache or persistently), the cache line can be written to memory (step (4)).

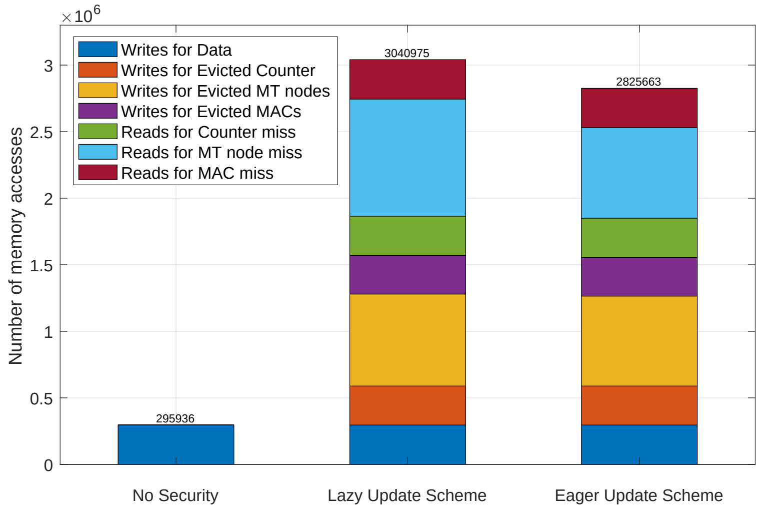


Figure 4.2: Breakdown of memory requests for flushing cache hierarchy in different scenarios (Total flushed cache blocks = 295936)

To better understand the performance implications of providing security during flushing cache hierarchy, Figure 4.2[1] compares the number of memory accesses incurred by flushing the cache hierarchy in a system without security, with two systems with security (eager update scheme or lazy update scheme for Merkle Tree). Note that even for merely flushing

[1]See the methodology section for more information about the assumed contents of the cache hierarchy upon crash. Note that EPD platform requirements must account for the worst case.

the cache content securely, secure EPD memory systems need 10.3x and 9.5x more memory accesses (hence more power hold-up budget) when lazy and eager integrity tree update schemes are used, respectively. We assume the cache hierarchy content before crash was randomly filled with sparse contents hence poor spatial locality in security metadata cache. Note that since EPD systems need to be designed with the worst case in mind, e.g., all cache lines are dirty in LLC for the case of non-secure EPD, we also need to consider the worst case for secure EPD and hence accounting for different access patterns. While we do not claim that randomly sparse cache hierarchy contents that are all dirty is even the worst case yet, we have observed an explosion in the required EPD power hold-up budget that is sufficient to motivate the need for novel solutions to enable secure EPD memory systems with reasonable backup power requirements.

4.2 Implementation

In this section, we describe a baseline support for secure EPD memory systems. After that, we discuss two different designs of Horus to guarantee memory security when there is a crash, but with minimal increase in EPD backup power requirements.

4.2.1 Baseline Secure EPD Systems

Since this is the first work to explore secure NVM in EPD systems, we start with defining a baseline implementation.

In secure but non-EPD systems (i.e., extremely limited or no backup power at all), the security metadata is updated in a persistent and recoverable way upon each write to persistent memory. To do so, schemes such as eager update (e.g., used in Triad-NVM(15)) always update the root of the tree before completing the data write. Hence, once the integrity tree is rebuilt, the system is considered recovered and can be verified using the up-to-date root on-chip. However, such a scheme can lead to high recovery time due to rebuilding the whole tree, and hence works such as Anubis(65) book-keeps which parts need to be rebuilt. On the other hand, while eager update is simpler to implement, lazy update scheme is generally faster as it only updates the leaf nodes on each write. Only upon the eviction of dirty nodes (including leaf nodes) do the updates propagate to the parent which will be placed in the cache and marked dirty. However, the lazy update scheme leads to crash consistency issues since the root will be stale upon system recovery. Prior works have presented mechanisms to enable recovery with lazy update scheme, however at the cost of extra complexity and memory writes(65; 4).

In non-persistent main memory systems (e.g., DRAM-based), either lazy or eager update schemes can be used without the need for any extra work to ensure recoverability. Secure EPD memory systems present an interesting design point where we aim for recovery-oblivious performance at run-time (due to sufficient power budget) but also need to flush sufficient amount of information upon detection of a crash to allow secure and consistent recovery.

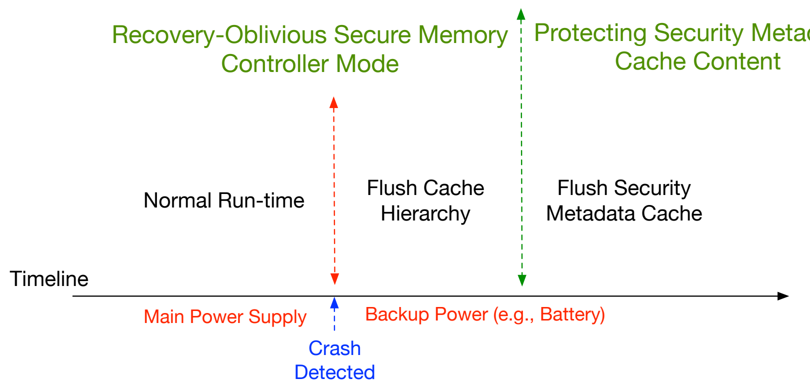

Recovery-Oblivious Secure Memory Controller Mode
Protecting Security Metadata Cache Content
Normal Run-time
Flush Cache Hierarchy
Flush Security Metadata Cache
Timeline
Main Power Supply
Backup Power (e.g., Battery)
Crash Detected

Figure 4.3: Timeline of events for in baseline secure EPD system.

Secure EPD Memory Systems: One possible design point is to use recovery-oblivious secure memory implementation, similar to those used with non-persistent memory. For instance, merely a lazy update or eager update schemes without any extra steps to ensure per write crash consistency. As shown in Figure 4.3, during run-time the EPD system can run with secure memory mode similar to non-persistent memory systems, i.e., recovery-oblivious, and hence minimal performance overheads. However, once a crash is detected, the EPD draining mode is triggered and an alternative power supply will be engaged (e.g., battery or capacitor). The first step is to write the contents of the cache hierarchy to memory while still using the recovery-oblivious secure memory mode. In other words, cache lines evicted from cache hierarchy will be encrypted and integrity-verifiable, however their most-recent security metadata might be not persisted yet (hence crash inconsistency). Thus, to solve the issue, once the cache hierarchy is flushed, the security metadata cache contents need to be either synchronized and flushed to their locations in memory or simply protected (e.g., using a small integrity tree) and flushed to a reserved region in memory. Leveraging EPD power, if eager update was used during run-time, then just flushing the security metadata cache to their original places will be sufficient. However, if lazy update scheme was used during run-time, then one way would be to scan through all metadata cache contents, bring their ancestor path (after verification) and propagate the update all the way through the root. However, this is costly, and hence an alternative approach is to calculate a single hash value over the metadata cache content, using a small integrity tree, similar to Anubis(65). Later, flush the metadata cache content to a reserved region in memory. Upon recovery, the metadata content will be restored from memory and verified.

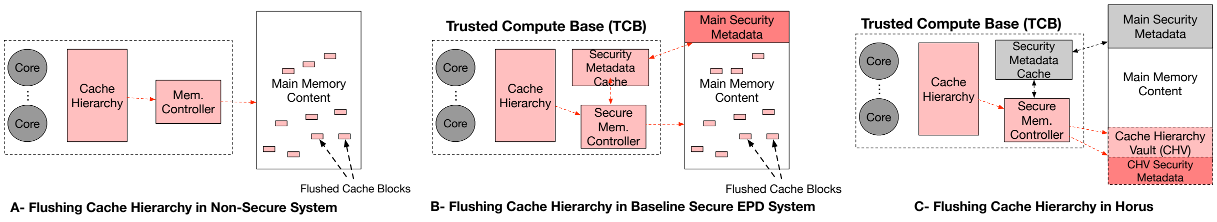


Figure 4.4: Comparison between the cache draining of (A) non-secure EPD system (B) Baseline secure EPD system, and (C) Horus.

Unfortunately, we observe that the step of flushing cache hierarchy can significantly increase the EPD power hold-up budget; as shown in Section 4.1, it requires more than 10.3x and 9.5x additional memory accesses when recovery-oblivious lazy and eager update schemes, respectively, are used. Note that if the cache hierarchy is needlessly flushed with recovery-awareness (e.g., Anubis(65)), then we would incur even more memory accesses to flush the cache hierarchy.

4.2.2 Horus

As discussed earlier, flushing the cache hierarchy while operating the secure memory controller in run-time mode can lead to significant increase in the number of memory accesses and operations. We also notice that such massive increase in the number of memory accesses results from security metadata fetches and updates, especially when the cache hierarchy before the crash was filled with dirty sparse cache lines, hence maximize the number of misses in the security metadata cache. Thus, our main ***design objective*** is to minimize the maximum number of extra operations needed upon flushing the cache hierarchy. Note that the power supply requirements are defined based on the worst case number of operations needed upon outage detection.

The first insight we leverage in Horus design is that in-place flushing of cache hierarchy contents is what leads to these extra accesses. Specifically, since flushed cache lines are written to their memory locations, then we need to use their address-specific metadata (e.g., BMT nodes, counters and MACs). However, sparse contents of cache hierarchy would naturally lead to many misses in such metadata. On the other hand, in-place updates without updating or using the respective (and verified) metadata would lead to security vulnerabilities (e.g., reuse of counter value) and/or functional errors due to the inability to discriminate which memory content was protected with the main BMT and their corresponding counters. Accordingly, we replace in-place updates when flushing the cache hierarchy with flushing the content to a small reserved region in memory which can be protected by using separate security metadata (i.e., CHV Security Metadata). We refer to this region by ***cache hierarchy vault (CHV)***, as shown in Figure 4.4. Note that in the baseline secure EPD system we might need to fetch security metadata from memory (in case of miss) since the counters to be used for encryption must be verified for integrity (as shown in Figure 4.4 part B). On the other hand, CHV's security metadata (shown in Figure 4.4 part C) is only written during the flushing stage and only brought back upon recovery to verify the contents. Later, in the next section, we will discuss why such counters used for encrypting CHV do not need to be integrity verified.

Our goal is to ensure that the flushed cache hierarchy content to CHV is ① protected in terms of confidentiality and integrity, ② such a protection is implemented using much less number of operations compared to in-place evictions/flushes, and ③ the flushed contents of the cache hierarchy are recoverable (and verifiable as guaranteed by ①). The rest of this subsection describes how each on of these goals is met through our Horus design.

Protecting Confidentiality and Integrity of CHV

The CHV should exactly contain all dirty cache blocks in the cache hierarchy at the time of the crash. Such a protection needs to guarantee the following: ① all dirty contents of the cache hierarchy are flushed to CHV. This implies the need for a mechanism to ensure that the number of blocks flushed upon crash detection is exactly the same as those existing in CHV. Otherwise, attackers can selectively omit new memory updates (which was present in cache) and hence replay a previous content. ② ensure that the addresses of the flushed cache blocks are also protected from tampering (including splicing attacks to swap the addresses of different cache blocks flushed to CHV). Note that the confidentiality of the addresses themselves is not protected since these in anyway will be observed later in the memory bus; access pattern leakage are beyond our threat model. However, their integrity protection is a must. ③ the confidentiality and integrity of the blocks written to memory. The cache blocks should be encrypted in a way that hides any temporal or spatial similarity between values as guaranteed during run-time. The temporal leakage could happen through observing the same ciphertext written for the same address or across different addresses over time (including cache draining episodes). In other words, we need to make sure that flushing the exact same content before encryption of the same address or other addresses even in different draining events will lead to different ciphertext written to memory. Finally, the integrity protection should ensure the cache blocks, as well as their addresses, are integrity protected and cannot be replayed from previous CHV contents of a previous draining process. In other words, we need to ensure the *freshness* of the CHV contents.

To this end, the CHV area includes drained **cache blocks** area, **addresses** of the drained cache blocks area, and **security metadata** area that stores the metadata used to protect the integrity and confidentiality of CHV. To provide such protections, one straightforward way is to treat the CHV areas to be protected as a miniature of the main memory, and hence have encryption counters dedicated for each memory block in the CHV area, and protect such counters using a small integrity tree rather than the larger integrity tree used for the main memory. Moreover, the protected CHV memory blocks have MACs co-located with that are calculated over the positional address in CHV, the encrypted memory block, and its corresponding CHV address encryption counter. In other words, a BMT style is used merely for protecting the address and data blocks in CHV. Unfortunately, this scheme requires fetching the CHV-address counters and verify them through the CHV integrity tree before using them, otherwise counter reuse with the same address could occur. Moreover, another limitation is the small BMT fetch and update process on each block flushing during the draining process.

Fortunately, we do not need to persist the CHV counters and a tree to protect them. Specifically, we observe that maintaining a monotonically increasing counter is sufficient to ensure unique initialization vectors for each flush operation upon draining. In particular, a persistent counter, always kept inside the processor chip, is incremented after each flush operation to memory. We refer to such a counter by ***drain counter (DC)***. By additionally book-keeping the latest number of drained blocks from the cache hierarchy persistently, we can know what is the most recent counter used for each flushed block. We refer to that register by ***ephemeral drain counter (eDC)***. The eDC value is cleared after each recovery of the system. By guaranteeing that each flush uses a unique initialization vector, we close both temporal and spatial leakage channels within a single draining process and across multiple draining episodes. Also, since the starting address of the CHV is fixed, we can relate each block flushed in the CHV to its drain counter value (address - CHV Base Address + DC - eDC).

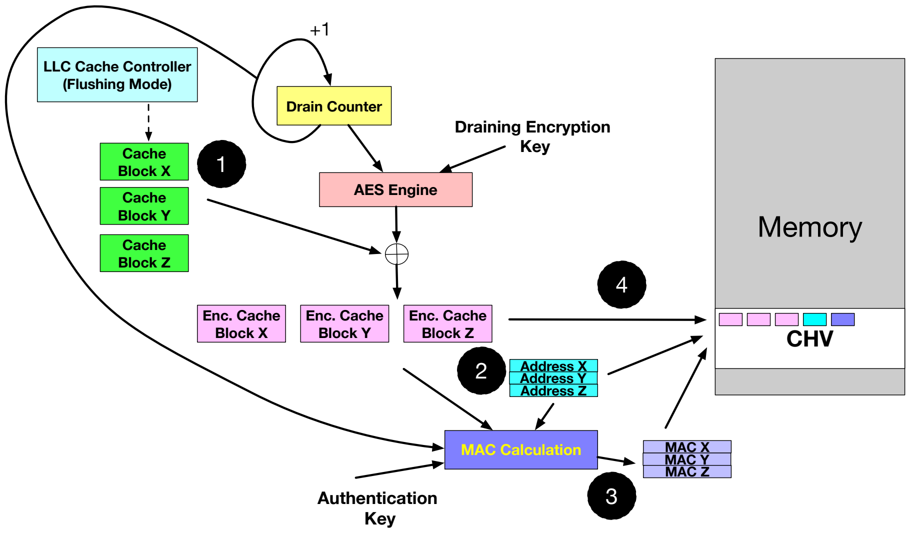


Figure 4.5: Simplified cache draining steps in Horus.

Figure 4.5 depicts the steps taken by Horus upon draining the cache hierarchy. As shown in Step ①, the LLC cache controller (or firmware loaded upon power outage) will start flushing the cache contents. Meanwhile, Horus encrypts these blocks using counter-mode encryption where drain counter used as initialization vector. In Step ②, the address of each block flushed is inserted to form a complete 64B block that just contains addresses in order, i.e., 8 addresses (assuming 64-bit addresses). In Step ③, MACs are calculated for each encrypted block along with its address (also stored in the address block) and the drain counter value used to encrypt it. Similar to addresses, MACs are combined into a single 64B and written to memory as one block. Finally, as shown in Step ④, the encrypted data blocks, address block and MAC block are written in order to the CHV. Note that in the figure we only show three data blocks being flushed, however these steps will repeat until all cache blocks are drained. For simplicity, we omitted the CHV address calculation part from the figure, however it can be simply calculated using the starting address of the CHV and drain counter as discussed earlier.

Write-Friendly and Optimized Protection of CHV

To further optimize the cache draining process, we coalesce the MACs calculated for multiple cache blocks (and address blocks) into one MAC block before writing it to the CHV security metadata region. Moreover, since the addresses of cache blocks are smaller than the memory write granularity, we coalesce multiple addresses into one block before writing it to memory. Moreover, to reduce the number of MACs need to be written for CHV integrity protection, instead of book-keeping a MAC value per memory block, we can do that for a coarser granularity. For instance, by merely maintaining two 64B MAC registers, we can use the first register to buffer 8 MAC values which once is full is used to calculate one 8B MAC value that gets buffered in the second register, before the first register is emptied. Once the second register is full, i.e., has eight MAC values, it will be written to memory as a single 64B block before being emptied. In other words, even though not to the extent of a full Merkle Tree, we hierarchically but efficiently (only using two registers) build two levels of MACs but only store the highest level to reduce the number of writes, as shown in Figure 4.6.

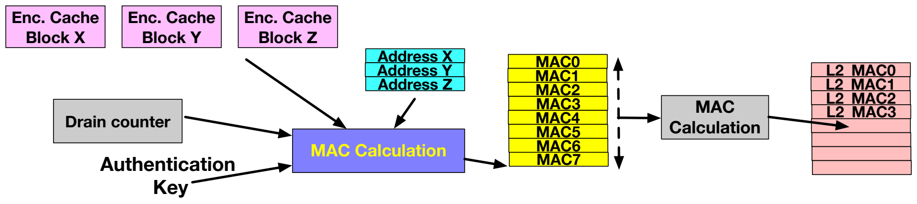


Figure 4.6: The Horus Double-Level MAC scheme for Horus

Our insight is that storing a MAC per memory block is mainly used to optimize the run-time memory access by avoiding over-fetching, since we do not need to bring neighbouring blocks to complete update/verification. However, in Horus, since it occurs only at draining time, we always have the neighbouring cache contents to be written to CHV. Note that neighbouring cache contents could be spatially far in terms of original memory location, however in CHV they are written contiguously. We dub this optimized scheme as ***Horus Double-Level MAC (Horus-DLM)***. On the contrary, we refer to the default scheme where a MAC is stored per memory block with ***Horus Single-Level MAC (Horus-SLM)***.

Recovery of Flushed Cache Content

The recovery process is straightforward. Upon power recovery, the contents in CHV are read back by the processor, in a reversed flush order, to recover the cache hierarchy. Data blocks and their corresponding address and MAC blocks, are read back together. The drain counter value used to encrypt each data block can be derived from its position in CHV along with the most recent value (after the crash) of the persistent *drain counter* on-chip. Upon the decryption and integrity verification of every data block drained to the CHV, we can either place them back in the LLC in dirty state or write them back to their original locations and update the main integrity tree accordingly (i.e., treat them as normal run-time writes). For simplicity, we opt for the first option that reads them back to the LLC and marks each as dirty. We assume the LLC cache controller will be aware that the system is in recovery mode and hence treats any fill operations as dirty blocks. However, if the cache is non-inclusive then to reduce complexity the second option (i.e., fetch, verify then write to memory using the main security metadata) may be used. Note that commercial EPD systems, e.g, eADR, already support flushing all caches in non-inclusive LLC systems, hence the only difference in Horus will be the recovery step in case we decide to place the flushed data blocks upon crash back to the cache hierarchy.

Security Analysis

The confidentiality of data blocks flushed to the CHV are guaranteed by ensuring a never repeating drain counter value, and hence unique one-time pads for counter mode encryption. Meanwhile, the integrity of the data blocks and their corresponding address blocks in the CHV can be verified using their MACs also stored in the CHV. Specifically, since the MAC of each data block is calculated using its address, the drain counter value used to encrypt it, and the encrypted data block itself, we can reproduce the MAC and compare it with the one inside the CHV. Thus, if the address block has been tampered with then the MAC generated using the tampered address will mismatch with the stored MAC, and thus will be detected. Similarly, if a data block has been tampered with, then the MAC will also mismatch and hence will be detected. Finally, if the attackers attempt to replay a previous CHV content or splice/swap current contents of the CHV, then the MAC will mismatch because the draining counter value used to protect data written to that location in the CHV will be different than the drain counter value used elsewhere.

4.2.3 Hardware Cost of Horus

In this section, we describes the hardware required by Horus. Horus leverages the existing secure memory support(AES, MAC) engines and secure metadata caches which are used during run-time, instead, Horus uses them during draining time. In addition, Horus needs some extra registers: two registers for DC and eDC (with simple ALU logic to increment the counter), one register for coalescing addresses, one register for coalescing MACs(two registers in the case of Horus Double-Level MAC shceme). In addition, Horus requires reserving a small region of NVM to be used as CHV. The area of CHV is nearly proportional to the cache size, $Size_{CHV} = 1.25 \times Size_{cache} + 1.125 \times Size_{metadata\_cache}$ in the case of Horus-SLM.

4.3 Evaluation

4.3.1 Simulation Setup

Table 4.1: Simulation Configuration Parameters

Processor	
Core	Single Core, X86, OoO, 4GHz
L1 Cache	2 cycles, 64KB,2-Way
L2 Cache	20 Cycles, 2MB, 8-Way
Inclusive LLC	32 Cycles, 16MB, 16-Way
DDR based PCM Memory	
Size	32 GB
Access Latency	read latency 150ns. write latency 500ns.
Secure Memory Parameters	
AES Latency	40 Cycles
Single Hash Latency	160 Cycles
Integrity Tree	a 10-level 8-ary Merkle Tree over NVM; a 5-level 8-ary Merkle Tree over secure cache
Counter cache size	256kB; 8-Way
MAC cache size	512kB; 8-Way
Merkle Tree cache size	256kB; 8-Way

To evaluate the performance of Horus, we use a cycle-level simulator, GEM5(17). As illustrated on Table 5.1, we simulate a single X86 core with 32GB DDR-based PCM. In Horus, the draining time is independent of the spatial relationships between the blocks in the cache hierarchy upon crash; however for the baseline, as also shown in the Section 4.1, it heavily depends on the spatial relationship between evictions as this highly impacts the behavior of the integrity tree cache. Since EPD battery requirements depend on the worst case, we assume a cache hierarchy content with extremely poor spatial adjacency between blocks. Specifically, we fill the cache hierarchy with cache blocks that are at least 16KB distant in their physical addresses; the 16KB was derived by dividing the simulated memory size by total size of cache hierarchy. While we do not claim that this is even the worst case, given the idiosyncratic behavior of the lazy update scheme with certain cache configurations, it is sufficient to demonstrate the high battery demands for a naive implementation of secure EPD systems. Meanwhile, Horus does not use integrity tree to drain the cache hierarchy and, hence, is oblivious to its upon-crash contents' spatial characteristics.

For evaluation, we compare the following four schemes: a baseline lazy update scheme (**Base-LU**), baseline eager update scheme (**Base-EU**), Horus with single level of MACs (**Horus-SLM**), and Horus with double-level MAC (**Horus-DLM**).

4.3.2 Overall Performance

The major components of the computing system, e.g., processor chip and memory modules, must be powered on until the system securely drains its cache hierarchy contents. Thus, a conservative proxy of the battery requirements in EPD systems is how long the system needs to stay powered on upon crash detection, hence using the alternative power source. Accordingly, we simulate the execution time starting from the detection of possible outage until the whole cache hierarchy is drain (including security metadata cache). Figure 4.7 shows the difference in execution time to drain the system.

As shown in the figure, both baselines using eager and lazy update schemes take 5.1× and 4.5×, respectively, longer time to drain the system compared to both Horus schemes. Accordingly, we expect Horus schemes to reduce the battery requirements for secure EPD systems by orders of roughly 4x-5x. Note that the ability to enable commercial EPD support, even without secure memory, is heavily restricted by the power supply power hold-up time capabilities. For instance, Intel's eADR cannot be enabled on systems with less than $10ms$ power hold-up time(32). Thus, reducing the hold-up time requirement for secure EPD systems by such significant amount will enable wider adoption of secure memory in future EPD systems. As also shown in the figure, Horus schemes reduces the draining time

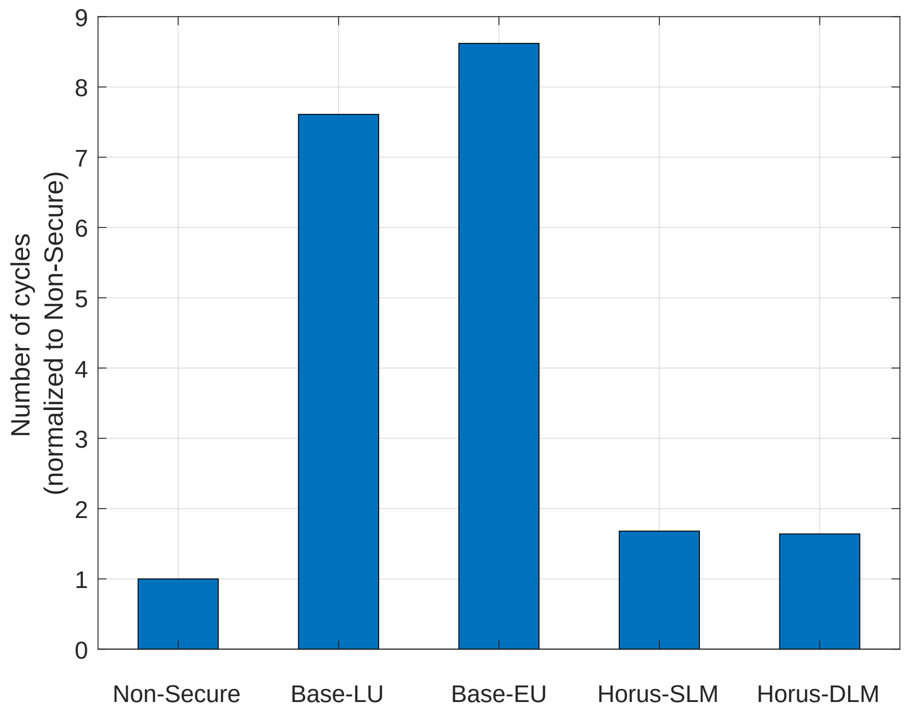


Figure 4.7: Normalized number of cycles

from 8.6× more time, hence 8.6× higher power hold-up time, to merely 1.7× compared to non-secure EPD system. As expected, the Base-EU takes the longest time to drain due to the large number of MAC operations and memory accesses (to fetch and update a whole integrity tree path on each write request).

4.3.3 Write Operations

To better understand the draining time reduction using Horus, Figure 4.8 shows the total number of memory write operations required for each scheme with a breakdown of the type of write request needed. We can observe that the majority of memory writes in the baseline are due to evictions of integrity tree metadata blocks due to security operations upon draining cache hierarchy. Both Base-EU and Base-LU use the main integrity tree upon draining cache, and hence poor spatial locality cache hierarchy contents can lead to massive number of misses in the BMT cache. We can also observe that, as expected, Horus-SLM incurs 8× smaller number of CHV MACs due to its coarser granularity protection. Also, a key observation from the figure is that flushing the metadata cache contents, once the cache hierarchy is drained, is negligible in all schemes. The reason is that the amount

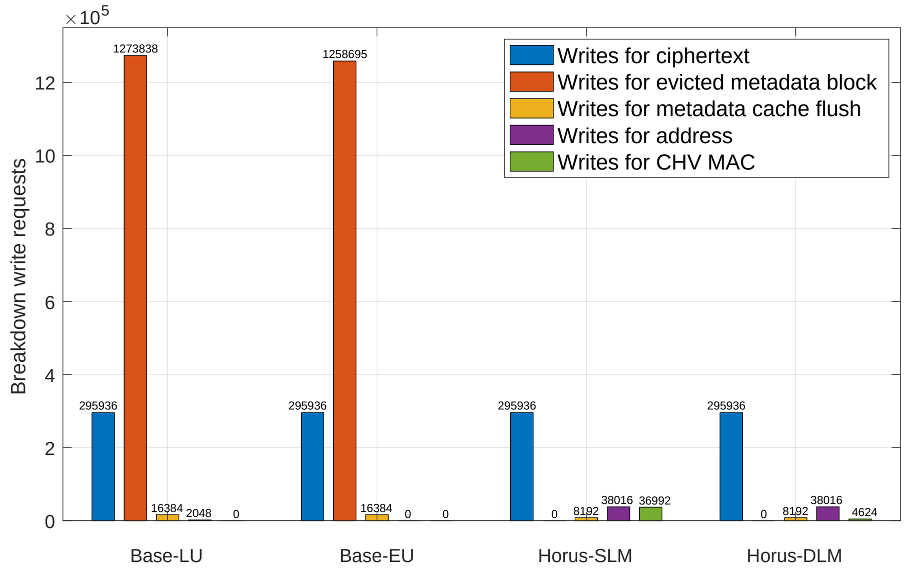


Figure 4.8: Breakdown of memory writes in different designs.

of metadata cache dirty evictions when draining the cache hierarchy is more dominant (in case of Base-LU and Base-EU), however even without such extra writes the number of cache blocks in LLC is much larger than metadata blocks. Thus, even in Horus, we can see that the number of metadata block flushes is negligible.

4.3.4 MAC Calculations

Another source of overhead during the system draining stage is the MAC calculations needed for authenticating the flushed contents. Figure 4.9 shows the breakdown of the number of MAC calculations needed for each scheme.

As shown in the figure, the eager update baseline (Base-EU) consumes the largest number of MAC calculations. The largest portion of MAC calculations in Base-EU is for updating the integrity tree (the second bar). On the other hand, since the integrity tree is eagerly updated, there is no need for any MAC calculations to protect the integrity tree (the fourth bar), but just merely flushing its content; the tree root is already up-to-date. On the other hand, for Base-LU, the largest contributor for MAC calculations is those used for verification, e.g., MAC to verify the counters and integrity tree nodes. Meanwhile, we can see that for both Horus schemes the most dominant MAC calculations are for the MACs to protect the flushed data blocks from cache hierarchy. Finally, we can observe that Horus-DLM consumes a 1.125× more MACs than that of Horus-SLM, mainly because of

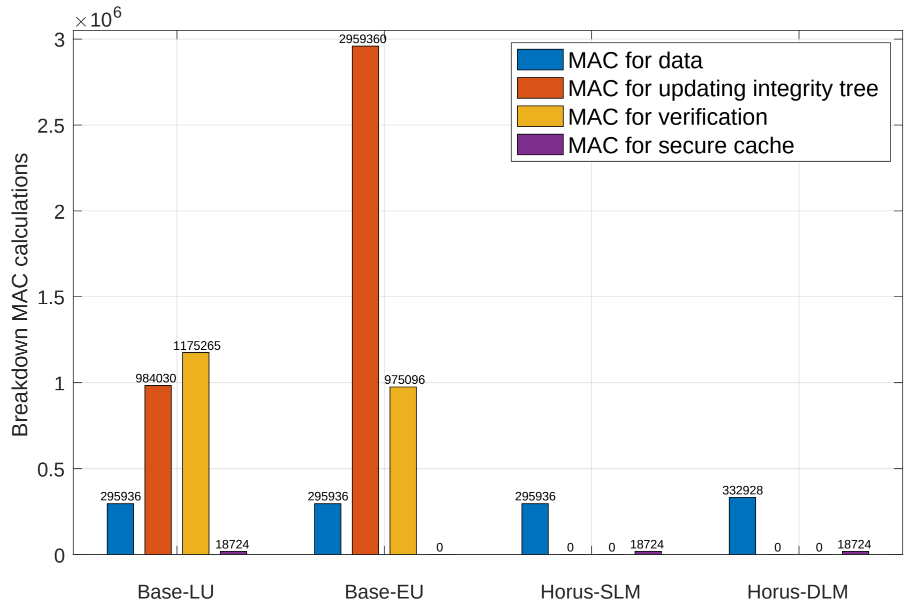


Figure 4.9: Breakdown of MAC calculations in different designs

calculating a second level MAC to reduce the number of MAC writes.

4.3.5 Sensitivity to Cache Size

In this section, we show the performance overhead of Horus on variable Last-Level-Cache sizes of 8MB, 16MB and 32MB. Figure 4.10 and Figure 4.11 shows the number of memory requests and MAC calculations normalized to Base-LU with corresponding LLC size. With LLC size of 8MB, 16MB and 32MB, both Horus schemes achieve at least 7.0x and 5.8x reduction in memory requests and MAC calculations, respectively, compared to Base-LU design.

4.3.6 Estimation of Recovery Time

In this section, we calculate the recovery time of Horus-SLM and Horus-DLM with varied LLC size from 8MB to 128MB. In Horus, the dominant parts of the recovery process are reading back the CHV content, integrity verification and data decryption. Figure 4.12 shows the recovery time of Horus-SLM and Horus-DLM. The parameters we use to estimate the recovery time are from Table 5.1. We can observe that even for LLC caches as large as 128MB, the recovery time of Horus-SLM and Horus-DLM is extremely small (0.51s and

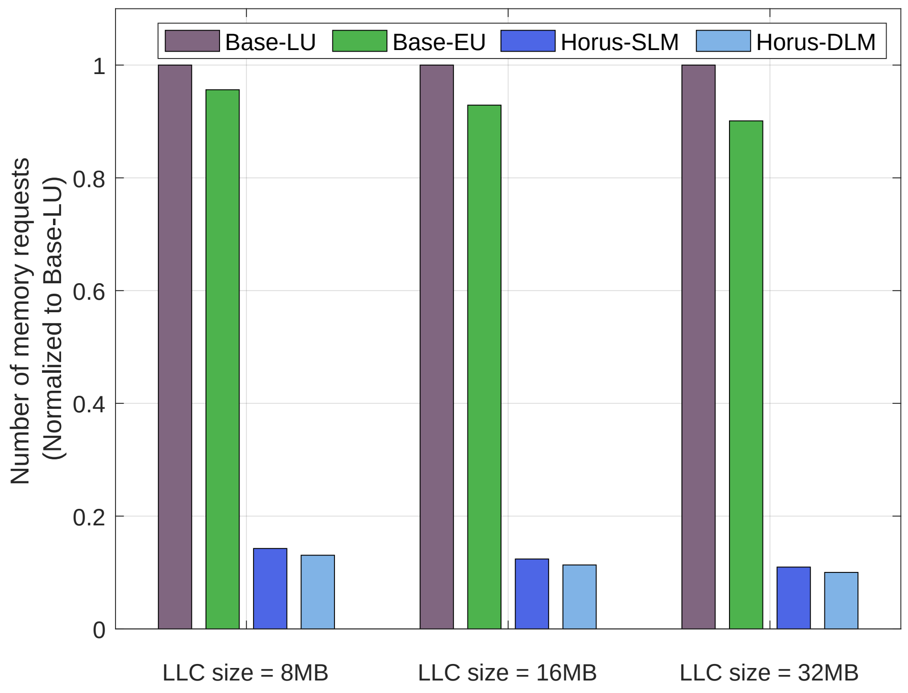


Figure 4.10: Normalized memory requests required in different Last-Level-Cache sizes of 8MB, 16MB and 32MB

0.48s, respectively). Hence, we believe that Horus can be used even in systems with very high availability requirements.

4.3.7 Estimation of Energy Costs and Battery Size

In this section, we evaluate the impact of Horus on energy cost during draining and the needed battery size. In eADR systems, once a power outage is detected a SMI interrupt will be signaled and a special code will be executed in the processor to flush the caches. Therefore, the processor needs to stay powered on during the whole cache hierarchy draining process. In our energy model, energy cost during draining mainly comes from 4 aspects: processor energy, NVM write operations, NVM read operations and secure operations. We use McPAT(37) to model the processor energy required during draining. We assume single NVM write and read operation takes the energy of 531.8nJ and 5.5nJ respectively(28). Energy cost for secure operations is minimal, compared with the other three aspects, therefore it is not included in our estimation. Table 4.2 categorizes the energy cost of the four designs. As shown in the table, energy cost during draining is dominated by processor energy, which is largely affected by draining time. Energy cost of Base-LU and Base-EU is 4.5x and 5.1x,

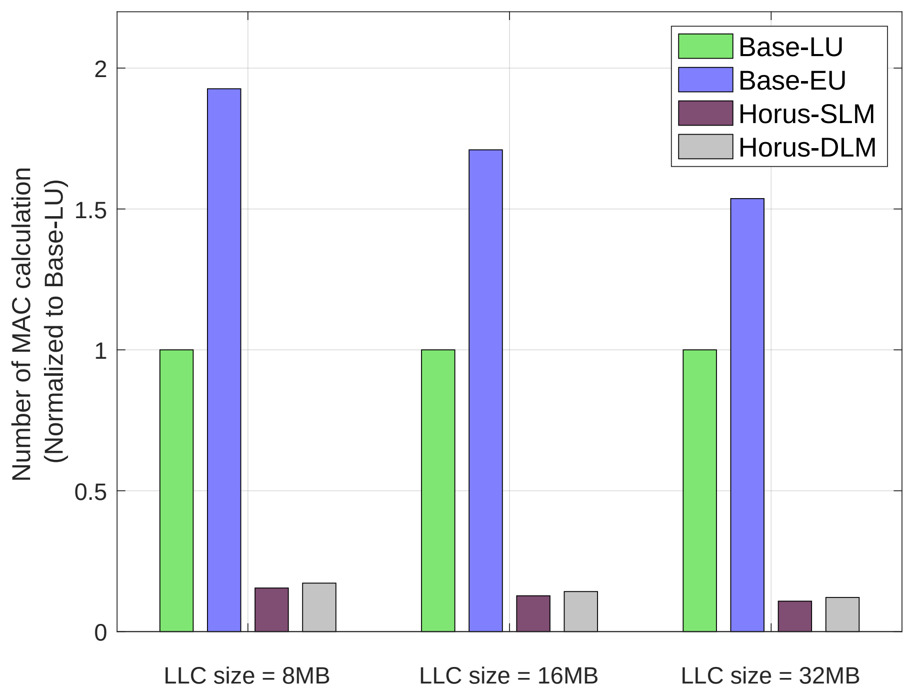


Figure 4.11: Normalized number of MAC calculations required in in different Last-Level-Cache sizes of 8MB, 16MB and 32MB

respectively, higher than both Horus schemes. Such required energy is closely similar to the draining time, because energy costs during draining is dominated by processor energy, which is mostly affected by draining time.

We use the energy cost in Table 4.2 to estimate needed battery size. Different battery technologies have different energy densities. In HORUS, we estimate two energy sources: super capacitors(SuperCap) and lithium thin-film batteries(Li-thin), using similar approaches as those used in BBB(3). The energy density for SuperCap and Li-thin is $10^{-4} Whcm^{-3}$ and $10^{-2} Whcm^{-3}$ respectively. Table 4.3 shows the battery size needed for the four designs using SuperCap and Li-thin. For both of the battery technologies, Horus reduces the battery size by at least 4.4×, compared with baseline design.

4.4 Related Works

In this section, we will discuss the related prior work on NVM system with battery-backed on-chip components and non-volatile caches (NVCaches). We will also discuss prior work on how secure NVM system adapts to system with on-chip persistent domain.

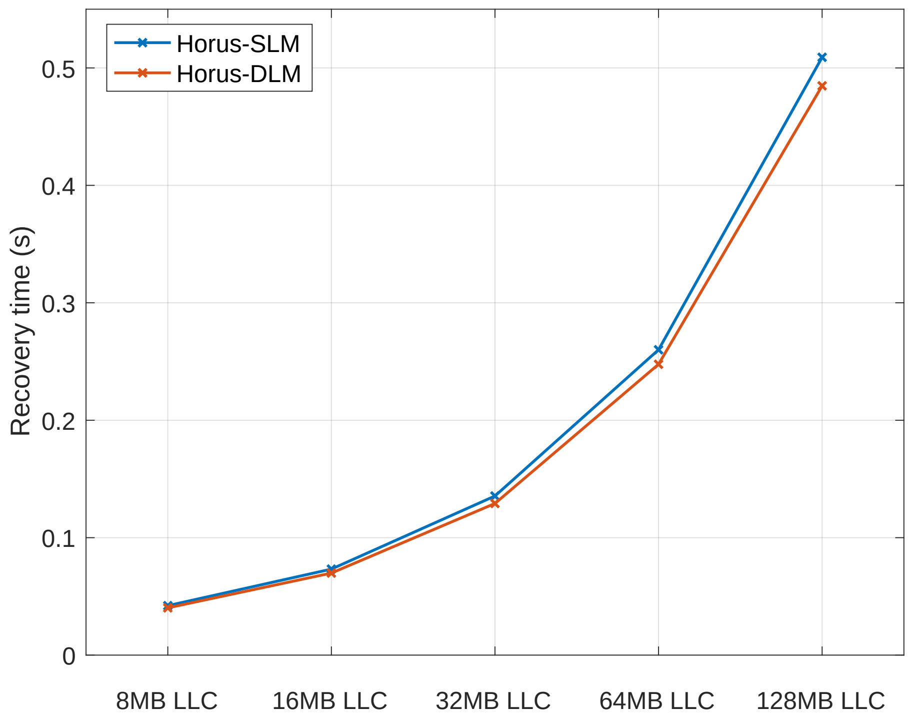


Figure 4.12: Recovery time of Horus-SLM and Horus-DLM.

Table 4.2: Estimation of Energy Costs of Different Operations during Draining

	Base-LU	Base-EU	Horus-SLM	Horus-DLM
Processor Energy(J)	10.21	11.54	2.25	2.20
NVM write operations(J)	0.84	0.83	0.2	0.18
NVM read operations(J)	0.008	0.007	0	0
Total(J)	11.07	12.39	2.45	2.38

Table 4.3: Estimation of battery size needed for draining

	Base-LU	Base-EU	Horus-SLM	Horus-DLM
SuperCap (cm^3)	30.7	34.4	6.8	6.6
Li-thin (cm^3)	0.31	0.34	0.07	0.07

NVM system with battery-backed on-chip resources: On-chip persistent domain is achieved by providing battery-backed on-chip components to flush the data to NVM during crash. For example, with battery-backed WPQ in memory controller (ADR solution(29)), persistent domain is extended to WPQ. With the entire cache hierarchy backed with battery(eADR solution(30)), persistent domain is extended to the cache hierarchy. BBB(3) proposed a battery-backed buffer attached with the L1 cache to hold the persistent data. Data in the battery-backed buffer is flushed to NVM when there is a crash. BBB extends the on-chip persistent domain to the same point as the eADR solution with smaller battery size. Unfortunately, none of the prior works addressed how secure memory can be implemented in such systems.

NVM system with NVCaches: Instead of using battery-backed cache, some work proposes using non-volatile cache (NVCache)(42; 51; 63). However, NVM technologies that are suitable for cache usage, e.g., Spin-Torque Transfer RAM (STT-RAM), have limited retention time that can vary from seconds to hours depending on other area/performance trade-offs(50). Thus, the majority of studies use it as regular cache without any persistence guarantees. Horus on the other hand aims to enable extending the persistence domain leveraging minimal back-up power.

Secure NVM system with on-chip persistent domain: Some works(27; 25) have researched how secure NVM system adapts to system with on-chip persistent domain. Dolos(27) assumes a secure NVM system with battery-backed WPQ. It proposes a Minor-Security-Unit to protect the WPQ to allow immediately flushing the WPQ content when there is a crash and also avoid causing large overhead on the performance of persistent application. Bonsai Merkle Forests(25) propose an on-chip non-volatile secure metadata cache for the integrity tree nodes. The single integrity tree is divided into small integrity trees by storing the roots of the small integrity trees in the non-volatile secure metadata cache to speedup updating the integrity tree. None of these prior works explore secure NVMs with on-chip persistent domain including battery-backed cache hierarchy. Unlike the prior works, Horus explores and provides solutions for how to protect the cache hierarchy when the battery-backed cache hierarchy is flushed to NVM during crash. Moreover, this is the first work that identifies the significant increase in the maximum amount of operations that need to be guaranteed power to securely flush the cache hierarchy.

4.5 Summary

In this chapter, we discuss the significant increase in number of memory operations, and hence draining time, for secure EPD. To mitigate the impact of such increase on the complexity and capabilities of power supplies required in future computing systems, we presented novel mechanisms to enable secure memory in EPD systems with reasonable increase in power hold-up time. Specifically, proposed Horus, which could effectively reduce the number of memory operations by at least 8×, and number of MAC calculations by 7.8×, compared to a baseline secure memory (using lazy update scheme). With these reductions, the draining time, and hence power hold-up time requirements, is reduced by 5× compared to the baseline secure memory scheme. Accordingly, Horus significantly reduces the required increase in power hold-up time for secure EPD systems.

CHAPTER

5

THOTH: BRIDGING THE GAP BETWEEN PERSISTENTLY SECURE MEMORIES AND MEMORY INTERFACES OF EMERGING NVMS

One requirement of secure persistent memory is crash consistency of security metadata. Previous works rely on ECC bits to enable fast persistence or recovery of security metadata. However, if such extra bits for ECC are not needed in future memory interface, the security metadata need to be persisted as separate writes. Specifically, ***if future memory interfaces do not have extra bits that are suitable for co-locating secure metadata with data, then there are no effective solutions for persistently secure NVMs. The only available solution is to incur separate security metadata writes with each persistent memory write in an atomic fashion***.

Emerging NVM modules, such as Intel's DCPMM modules, compute ECC bits internally and attempt to correct them before reporting errors (interrupt) to the host(1; 34). Such

DIMM internal implementation of ECC is likely to dominate the memory industry due to the following: (1) many memory technologies with different reliability (and hence ECC algorithms) are available, and thus relying on the processor-side memory controller to implement them is infeasible. (2) Leaving internal ECC implementation details to memory vendors allows more flexibility in choosing suitable ECC memory for critical domains (e.g., safety-critical systems) while more relaxed (or absence thereof) ECC in less critical domains. (3) With certain modules supporting encryption internally, the best place to calculate ECC is after the encryption is done, otherwise the errors diffuse after decryption and makes it difficult for host-level ECC to fix it. For instance, Intel's DCPMM internally leverages AES-XTS and hence if there is no strong ECC support inside the module then errors are further exacerbated when decrypted internally before being sent to the host[1]. Moreover, the granularity of AES-XTS cipher blocks is 128 bits and hence even a single bit error within a 128-bit of the memory block can turn a 128-bit into a nearly-random value after decryption inside the DIMM; making it challenging to fix at the host side even with strong ECC support. Similarly, future NVMs (e.g., SK Hynix's 3DVXP) are envisioned to be interfaced through computer express link (CXL) memory semantic protocol(53), where the width is 66B of which only 2B are used for ECC of the transmission, and the remaining 64B is the payload (e.g., cacheline).

In today's state-of-the-art secure NVM implementations(60; 65; 19; 58; 27; 59; 49), the message-authentication codes (MACs) of data is assumed co-located with data using additional pins, while the encryption counter used to encrypt the data is persisted through overriding (or repurposing) the ECC bits. Without the need for host-side ECC, which could be replaced by on-DIMM ECC in future interfaces, such implementations may lose their effectiveness. Specifically, they will require two additional writes for the MAC and counter blocks. Such overheads are unacceptable in terms of both NVM lifetime and performance. Due to storage efficiency, encryption counters and MACs are not co-located in the same memory block; encryption counters has much less storage overhead compared to MACs (typically 12.5% for MACs vs. 1.56% for counters), and hence they are separated in different blocks, which causes two extra separate block writes to memory for each memory block write.

We observe that much of the write amplification in secure NVM occurs because of the disparity between counter and MAC sizes and the write granularity to memory. On a

[1]Unlike AES-CTR mode which is generally used in the processor side for confidential computing, AES-XTS directly feeds the ciphertext as input to the AES algorithm to complete decryption, and hence significantly diffuses any bit errors upon decryption.

memory write, an additional 8B MAC and 8B counter (minor + major counters(56)) are updated along with the data, and they are held in separate blocks in memory. Because only one counter or MAC in each block is updated, we refer to these as a partial updates. When partial updates are persisted to memory to maintain crash consistency, both full blocks (of the counter and MAC blocks) need to be persisted causing the write amplification[2].

Our main insight is that we can pair a large off-chip persistent buffer for partial updates with the normal write-back behavior of our secure metadata cache to create a new write efficient architecture. Partial updates are packed together efficiently and written into the persistent buffer to provide crash consistency while the secure metadata cache continues operate as a write-back cache. We observe that if we buffer partial updates for long enough, when they are evicted from the persistent buffer, no additional writes are needed because the state they hold has already been written to memory efficiently through other means. For example, the secure metadata cache will eventually perform the update through its natural write-back process, likely after accumulating many updates to the same entry saving many writes. Another example is if another yet younger partial update is made for the same metadata (i.e., MAC or counter), then all older partial updates to the same metadata are stale and can be discarded, saving those writes. Moreover, if the metadata cache block is persisted for any reason, then all partial updates for that same metadata block can be safely discarded because the metadata block in memory is already up-to-date. With these insights, we conclude that a large persistent buffer can provide lots of opportunities to avoid writing partial updates to memory. Most importantly, we observe that the overheads for maintaining such a persistence buffer are minimal because we do not need the persistent buffer to reside on-chip. Many partial updates can be packed into one memory block and written to memory together. Thus, with relatively minimal memory buffering overheads, we significantly increase the probability of eliminating the full-block write upon each partial update.

In this chapter, we present ***Thoth***, which ensures crash-consistency while exploiting temporal and spatial combining of partial updates in a secure and elegant manner. Thoth aims to (1) realize a persistent ***partial updates buffer (PUB)*** in memory (2) upon eviction of a partial update entry from the PUB, discard the write-back of the corresponding metadata block when no longer necessary (i.e., updated later or evicted from metadata cache in processor).

[2]The write amplification further grows with NVMs that use larger access granularity (128B or 256B in Intel's DCPMM(55)) while the MAC and counter sizes remain the same.

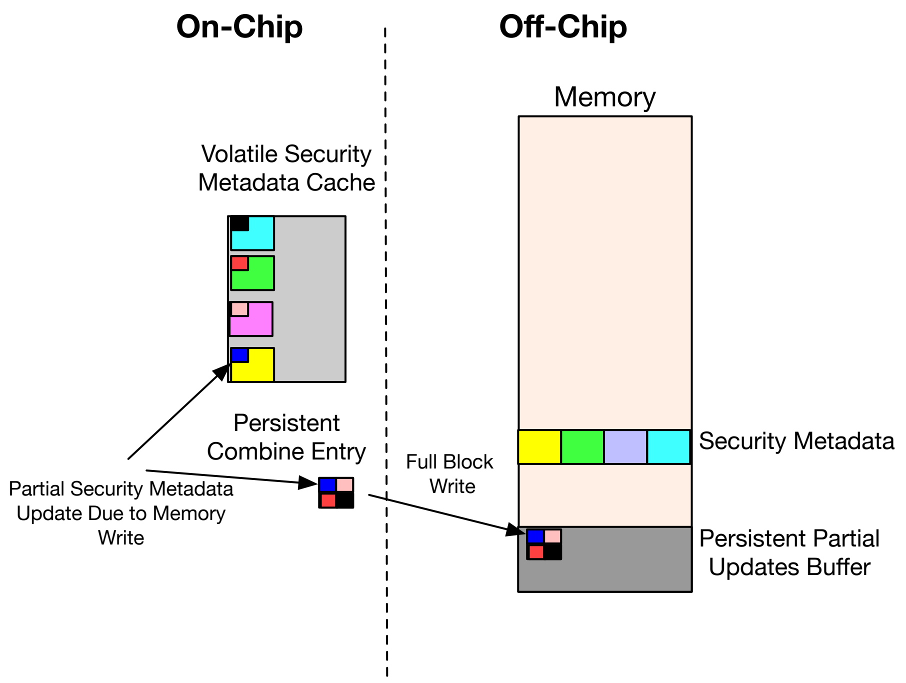

On-Chip
Off-Chip
Memory
Volatile Security Metadata Cache
Persistent Combine Entry
Partial Security Metadata Update Due to Memory Write
Full Block Write
Security Metadata
Persistent Partial Updates Buffer

Figure 5.1: High-level example of partial updates buffer (PUB) system-level layout.

5.1 Motivation

In this section, we demonstrate the major observations we have about the effectiveness of long-term buffering of partial updates, which will later motivate our design decisions for Thoth. First, we define *partial security metadata block updates* as updates to a specific MAC or encryption counter within a MAC cache block or counter block. Such partial updates occur due to writing the memory block protected by these MAC and counter. However, due to the access granularity in memory, MACs, and similarly counters, are grouped into memory blocks that are fetched and cached together (e.g., in 64B block granularity for conventional DDR). Thus, upon an update to a MAC or a counter block, the whole blocks that contain them will be persisted to NVM in addition to the data block. Whole block persists triggered by partial security metadata block updates are key source of write amplification for secure NVMs and we want to avoid as many as possible to enable greater compatibility with emerging memory interfaces.

We imagine a new organization for crash consistent NVM that is able to efficiently reduce write amplification. As shown in Figure 5.1, we add a partial updates buffer (PUB) in memory to collect these partial updates. As shown in the figure, we still use a write-back security metadata cache that is updated on each security metadata update. However instead of also persisting the full metadata block to memory due to a partial update, we combine these partial updates, pack them tightly into blocks, and buffer them in memory at a fraction of the number of writes they would otherwise cost. Note that during run-time, all metadata will be strictly fetched and evicted through the normal path and not through the PUB, e.g. we check for it in the cache, and otherwise bring it from its *original location* (not the PUB). Also, upon eviction from the secure metadata cache, if dirty, then it needs to be written to its original location. However, upon crash events, the most recent update in the volatile secure metadata cache might be lost, and thus we need to recover the partial updates from the PUB. This architecture may at first appear to increase the number of writes to memory since evictions of the partial updates from the PUB would ultimately require full block persists, however, this organization actually significantly reduces writes.

Our ***key observation*** that enables this reduction in writes is that the vast majority of partial security metadata updates when evicted from the PUB need no additional writes, if they are persistently buffered for long enough. This is because, after a long enough time passes, the probability is high that memory already contains their update. The following reasons explain why the probability is so high. (1) While the partial update is buffered, the security metadata block which the partial update belongs to may have already been evicted from the secure metadata cache and persisted to NVM; such a probability increases over time due to the natural changes in the working set of applications over time. (2) An older partial update eviction may have already caused a full security metadata block persist which included the partial update from the current entry (already in the persisted cached metadata block). Note, this case captures spatial locality of updates to metadata within the same block. (3) A younger partial update arrived to the same location as an existing partial update. In this case, any older partial update for the same metadata can be discarded because it is stale. This case captures the temporal locality associated with a data block that is frequently updated, a common trait in persistent applications.

We observe that the aforementioned three cases are very common and their probabilities increase significantly with the increase of the partial buffer size. Figure 5.2 breaks down the cases we observe upon eviction from a hypothetical FIFO partial buffers with sizes of 500,000 entries (A), 5,000 entries (B), and 50 entries (C). The ***written-back*** percentage represents the probability of upon the eviction of a partial update its security metadata cache block still needs to be persisted to memory. Meanwhile, ***already-evicted*** represents the case of when upon a partial update eviction from the buffer its up-to-date metadata block has been already evicted from cache and written back to memory, and thus the

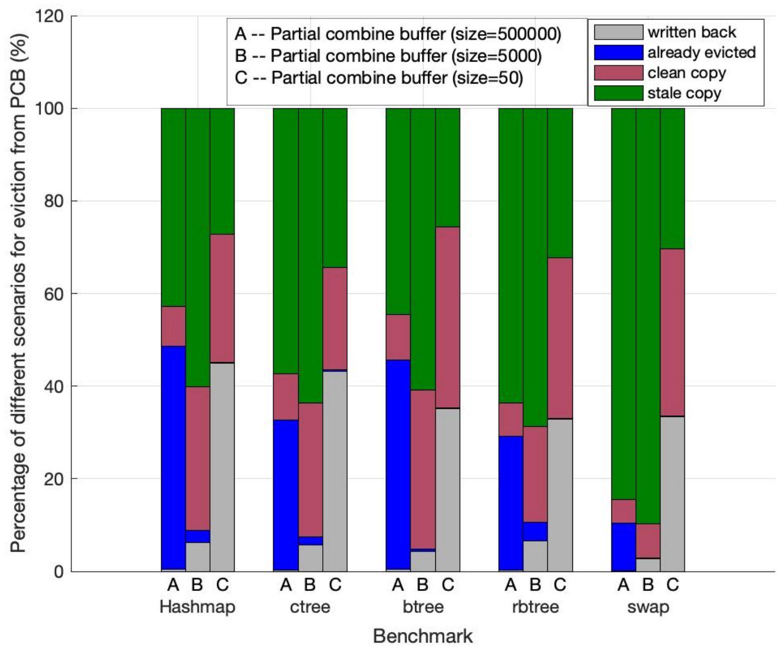


Figure 5.2: Breakdown different scenarios for evicted copies from Partial Combine Buffer (PCB).

partial update can be discarded safely. The ***clean copy*** case is when a partial update is upon eviction, its metadata block is still in the metadata cache but in clean state, which means it was either persisted due to partial update eviction or evicted from cache then fetched again later. Finally, the ***stale copy*** indicates the case where upon the eviction of a partial update its metadata block was in the cache and dirty, however the partial update is stale, i.e., a newer partial update was inserted in the PUB. Thus, the stale partial update can be safely discarded.

As shown in Figure 5.2, for all the benchmarks we can observe that with a large enough partial buffer size, the majority of partial updates (99.5% on average for the 500,000 buffer) do not cause a full block persist upon eviction. Rather when the larger PUB sizes are used, the common case is that evicted entries are stale, and the second most common case is that the partial update has already been evicted and written back from the secure metadata cache. In neither case is it necessary to write the evicted PUB entry back to its original location in memory to ensure crash recoverability.

Based on these insights, we observe the importance of large PUB and their great potential for eliminating extra security metadata block persistence operations that ensure crash consistency. In short, our approach replaces the certain extra security metadata writes with a very small probability of an extra write plus the low overhead of buffering packed partial updates off-chip. The rest of the chapter demonstrates how to realize this hypothetical design in an elegant yet highly effective fashion.

5.2 Implementation

In this section, we describe the design options and rationale behind the various design choices for Thoth.

First, let's more formally define the ***partial updates buffer (PUB)*** alluded to earlier in Section 5.1. A PUB contains partial security cache block updates, i.e., only the portions to be updated for security metadata blocks. For instance, a memory write would cause updating the MAC block containing the MAC and the counter block containing the counter correspond to the memory location to be written. However, the partial updates are the MAC (8B field) and the impacted counter (typically just the 7-bit minor counter). Thus, a PUB should merely contain the new values of the impacted MAC and (minor) counter. To ensure correctness we need to ensure the following: ① the partial updates need to be securely recoverable and protected. ② the volatile security metadata cache should be updated with the most recent metadata (e.g., MACs and counters) upon buffering partial updates to the PUB; this ensures consistency. ③ there should be a safe handling eviction policy when the PUB fills up.

As shown earlier, in Section 5.1, the PUB needs to be quite large (e.g., 500,000 entries) to maximize the benefits. Thus, the logical place to place the PUB is off-chip. However, this brings us to our first design question, how do we arrange partial updates in memory?

5.2.1 PUB Organization

Since the memory write granularity is at a cache block granularity (e.g., 128B or 256B), we need to *persistently* pack partial updates into blocks before writing them back to memory. The buffer itself is managed as a FIFO circular buffer where two counters are used, one to indicate the start and one to indicate the end. A third register is used to indicate the base address of the buffer. Once the start equals the end, no more insertions are allowed until evictions occur (and hence the end variable is incremented mod the buffer size).

To persistently hold partial updates as they are forming a full cache block to be written off-chip to the PUB, a few entries are reserved from the processor's internal WPQ which is

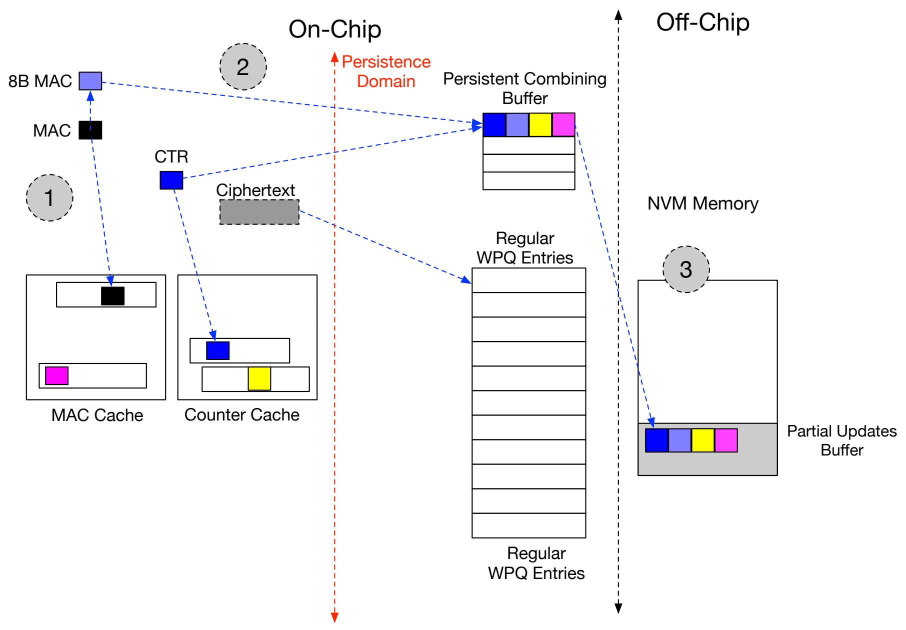


Figure 5.3: Overview of the coalescing steps leveraging reserved WPQ entries.

battery-backed through the ADR support(29).

As shown in Figure 5.3, multiple partial updates can be combined together in a single WPQ entry. When a new memory write occurs, after obtaining the verified counter value and calculating the new MACs, calculating the new integrity tree root and nodes[3], the new ciphertext and updated security metadata need to be persisted before the transaction is considered complete and persistent. Since an 8-to-1 MAC is computed over the ciphertext, a 16B MAC is computed for 128B cache block and a 32B MAC for 256B. To be able to pack more partial updates in one WPQ entry, an 8B second-level MAC is computed and allocated in PUB. As shown in Step ①, a specific ciphertext and its accompanying security metadata (MACs and counter) are ready to be persisted, and the counter and first-level MAC can be made to caches. In Step ②, which can be overlapped with ①, the partial updates (counter and 8B second-level MAC, not their full cache blocks) are placed in a reserved WPQ entry, while the ciphertext block is inserted in the regular WPQ. Now as the reserved coalescing WPQ entry is full, it can be written to the PUB as shown in Step ③. We omitted the steps for updating the bounds of the buffer upon insertion for the sake of simplicity.

Each PUB cache block is comprised of 9 partial updates (for 128B cacheline size) or 19 partial updates (for 256B cacheline size).A partial update entry contains the {address, MAC,

[3]Note that this also can be done lazily by securely tracking the cache content as done in Anubis(65) and Phoenix(4).

counter, status}. The *address* is 32b and represents the cache block address which the MAC and counter correspond to, this can address address up to a 512GB module (note that there are unused bits that can be used for address if larger modules are used in the future). The *counter* is the 7b minor counter; we persist the counter block immediately when a minor counter overflow occurs (i.e., the major counter changes). The MAC is 64b for both designs with 128B cache blocks and 256B cache blocks. Finally, the *status* bits (2b) are used to help on deciding the actions upon the eviction of this partial update entry from the PUB (will be discussed later in this section). To eliminate issues with crashes while the coalescing entry in WPQ is not full yet, we duplicate the existing partial entries upon a crash to fill a full cache block.

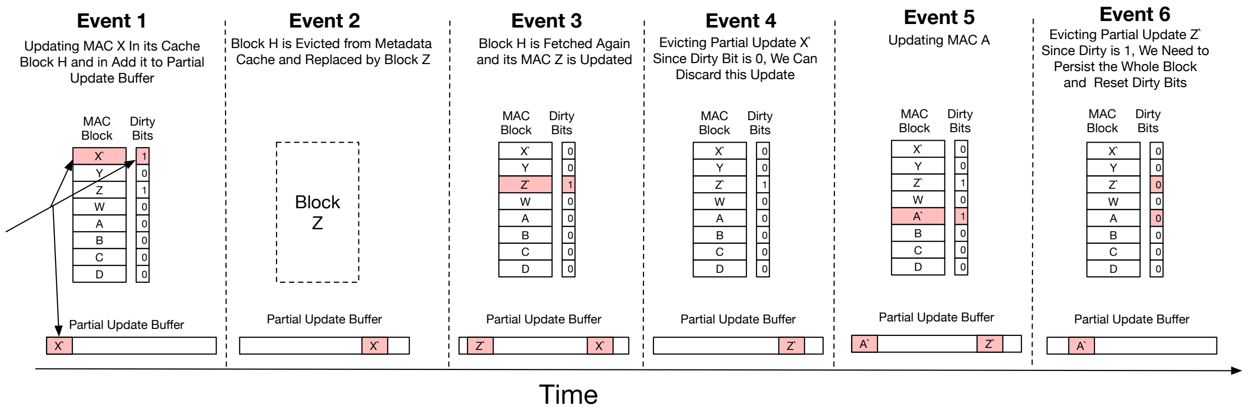

Event 1
Updating MAC X In its Cache Block H and in Add it to Partial Update Buffer
Event 2
Block H is Evicted from Metadata Cache and Replaced by Block Z
Event 3
Block H is Fetched Again and its MAC Z is Updated
Event 4
Evicting Partial Update X` Since Dirty Bit is 0, We Can Discard this Update
Event 5
Updating MAC A
Event 6
Evicting Partial Update Z` Since Dirty is 1, We Need to Persist the Whole Block and Reset Dirty Bits
MAC Block
Dirty Bits
Block Z
Partial Update Buffer
Time

Figure 5.4: Demonstration of how WTBC captures the various PUB eviction scenarios. MAC block is shown, but similar events and actions would also occur for counter block.

5.2.2 PUB Eviction Mechanism

The second question we aim to answer is: how to efficiently implement an eviction policy from the PUB? Note that this is perhaps the most critical aspect in Thoth; without an efficient mechanism, most partial updates will end up eventually causing their corresponding metadata blocks to be persisted when these partial update entries are evicted from the buffer. Upon an eviction of an entry from the PUB, it is time to decide whether to persist the partial security metadata blocks' updates to their original locations or not. To implement this, upon an eviction from the PUB entry, we fetch the victim (e.g., last) partial updates memory block from the PUB, which contains a pack of updates. For each entry, we check

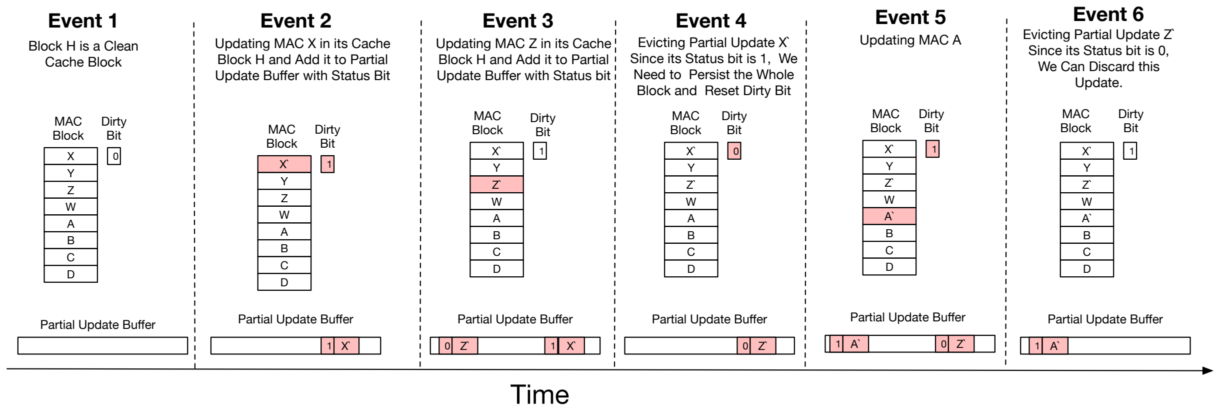

Event 1
Block H is a Clean Cache Block
Event 2
Updating MAC X in its Cache Block H and Add it to Partial Update Buffer with Status Bit
Event 3
Updating MAC Z in its Cache Block H and Add it to Partial Update Buffer with Status bit
Event 4
Evicting Partial Update X` Since its Status bit is 1, We Need to Persist the Whole Block and Reset Dirty Bit
Event 5
Updating MAC A
Event 6
Evicting Partial Update Z` Since its Status bit is 0, We Can Discard this Update.
MAC Block
Dirty Bit
Partial Update Buffer
Time

Figure 5.5: Demonstration of how WTSC captures the various PUB eviction scenarios. MAC block is shown, but similar event and actions would also occur for counter block.

to see if we actually still need to persist that partial update or not, i.e., if it has been already persisted through the natural cache writeback path, persisted due to a later partial update to the exact same MAC/CTR, or simply a prior PUB entry that falls within the same security metadata block has been evicted, and hence caused the spatially-shared security metadata block in cache to be persisted.

Security Aspects: The first aspect that arises here is whether or not we need to verify the integrity of the partial updates entries once they are read back for eviction from the PUB (and potentially persist their corresponding security metadata). Since the most recent counter/MAC values are also either in the volatile counter/MAC cache or already evicted to memory, the partial update value is never needed or incorporated in any update during run-time and normal operation. In other words, during normal execution time, original location of secure metadata blocks in NVM is always updated by the copies in secure metadata cache. However, upon a crash, the most recent values of these updates in the cache are lost, and thus we need to recover them through reading the PUB. Fortunately, since the most recent counter values are already incorporated in the integrity tree root persistent inside the processor(12), or a secure and protected shadow cache in memory (as in (65)), we can verify these partial updates upon secure reconstruction. In other words, during crash, tampered PUB is detected using the integrity trees. For example, tampering a most-recent counter value that is lost in the cache is detected by leveraging the root calculated over secure metadata cache. Tampering a counter value that is already updated in-place before crash will result in failure in reconstruction of the main integrity tree. Thoth only replaces encoding MAC/CTR blocks using extra bits at the memory bus with a temporary combination buffer before they end up being updated in place.

Detection of Stale Partial Updates: Another major aspect is how to efficiently detect if a partial update is no longer needed, e.g. due to the cache eviction of the security metadata cache block already containing this update to memory. As mentioned earlier, we aim to identify the following three cases: ① the up-to-date cache block containing the partial update has been already evicted from cache. Note that this case should also incorporate the scenario for evicting the cache block after the partial update occurred, but it has been read later and other parts of the block were updated. ② the same partial metadata has been updated later and added to the partial buffer ③ a prior metadata block persist operation, e.g., due to eviction of a PUB entry, caused other partial updates that share the security metadata block to be already persisted along with the block. To allow detecting these cases, we start with a simple design, which we call **Write-Back Through Bitmask Checks (WTBC)**, that relies on fine-granularity tracking of dirtiness of counters/MACs within security metadata blocks. Upon a fetch of a security metadata block, all dirty bits for all its MAC/CTRs are set to 0. Only upon a partial update to any of them will that specific MAC/CTR's corresponding dirty bit be set to 1. Also, upon persisting a metadata block, all of its dirty bits are reset to 0. Figure 5.4 depicts scenarios that reflect how WTBC captures cases ① and ③, in Event 4 and Event 6, respectively. Unfortunately, merely relying on the dirty bits to capture case ② is insufficient; if any write to the same data block occurs between the evictions of two prior partial updates correspond to the same data block, then the later eviction would still cause a persist of the metadata block as the dirty bit will be one. However, the first partial update eviction could have already caused persistence of the block which potentially contained the later partial update. One more efficient way to capture case ② is to compare the value to be evicted from the PUB with the verified value in the metadata cache. Based on our observation that each partial update for MAC/CTR will generate a new unique value; thus, by comparing that value along with the dirty bit value, we can know if the partial update is stale or not.(Note, evicted partial update's MAC needs to be compared with a second level 8B MAC computed over the corresponding MAC in the secure metadata cache.) In other words, the partial update needs to occur only if the dirty bit is 1 and the partial update's MAC/CTR value is different than that in the cache block. Note that this is also safe to do since we merely use the partial update value to decide to persist or not, however the newest counter value is already incorporated in the integrity tree (root). The main issue of the WTBC scheme is its need for fine-grain dirtiness tracking of security metadata, which can add extra storage overhead to caches.

To avoid adding the area overhead, we propose our second design, which we call **Write-**

Back Through Status Checks (WTSC). WTSC is based on our observation that upon the insertion of an entry to the PUB, we can use the block dirty status to determine if the partial update value is captured during the eviction of an older partial update entry or not. Specifically, if the metadata block is already dirty upon a new partial update, it means that a prior partial update occurred and has not been evicted yet (otherwise the block would have been cleaned upon its persistence). We observe that upon fetching a metadata block from memory, only the first partial update which converts its status to dirty needs to persist the block upon the partial update's eviction. Meanwhile, for all later partial updates arriving while the block is already dirty, their values will be captured and persisted implicitly upon the persistence of the partial update preceding them which caused the block to become dirty. Thus, we record the dirty bit status upon a partial update along with its entry in the PUB, which we refer to as the *status bit*. However, WTSC partially captures case ① and ②; if a another (or same) metadata partial update within the same data block occurs after eviction of the block, then the status bit will be set to 1 in the corresponding entry. However, upon the eviction of a the later entry, we will needlessly still persist the block even though the new partial update value might have been already captured. Thus, WTSC captures all the cases needed for functional correctness but is more conservative as it fails to precisely detect if the persist is not needed (not the other way around) compared to WTBC. Figure 5.5 demonstrates how WTSC captures the various scenarios.

Fortunately, we empirically found that WTSC, even though an approximate but needlessly more conservative version of WTBC, is sufficient to eliminate most of the writes upon the eviction of partial updates from the buffer. Thus, for the rest of the work, we select WTSC as the eviction persist policy of partial updates from the PUB.

5.2.3 Interactions with WPQ

Today's processors support a small persistent write buffer on-chip called the write pending queue (WPQ)(29). The WPQ is typically a small buffer (e.g., 64 entries) and merely holds the evicted blocks before being written to NVM; this reduces the latency it takes to persist data compared to waiting until the block reaches the NVM. Adding the PUB raises the design question of how to interact and leverage the WPQ. First, to persistently combine partial updates, we simply dedicate entries from the ADR-backed WPQ to combine partial updates together. We refer to the WPQ entries used to coalesce independent partial updates as the *persistent combining buffer (PCB)*, and the rest of the WPQ entries as *WPQ*.

Logically, there are two ways to arrange the ordering between PCB and WPQ, PCB-

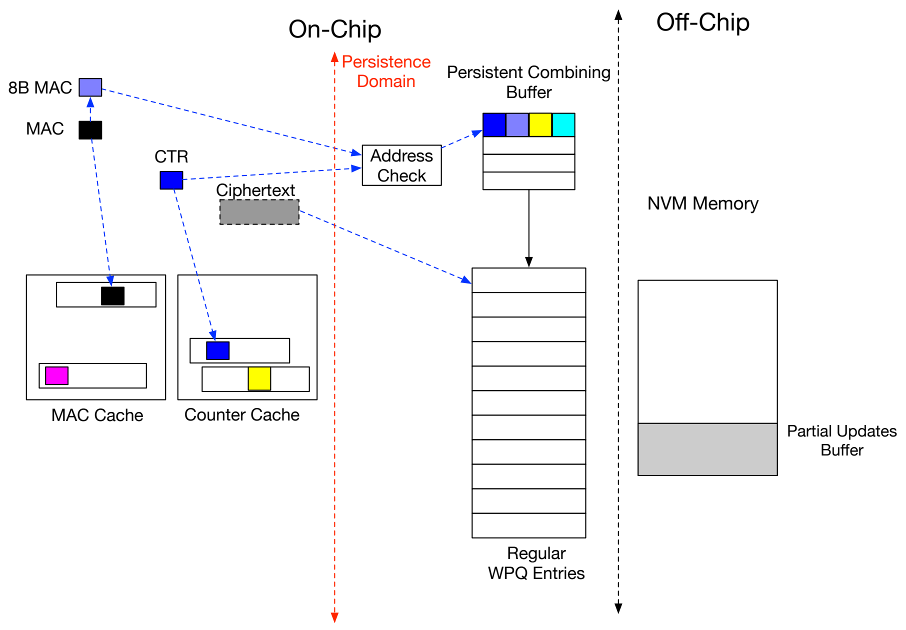


Figure 5.6: Overview of PCB-before-WPQ approach

before-WPQ or WPQ-before-PCB. The former waits until the PCB is full before placing the entries in the WPQ using the address of where this block needs to be written in the PUB in memory. However, this approach misses opportunities to coalesce updates to the same security metadata block once it is evicted from the small size PCB; even though partial updates could belong to the same security metadata, they will look like independent entries once inserted in the WPQ (they are tagged with different addresses in WPQ).

The other approach is to place the PCB after the WPQ (PCB-after-WPQ approach). In this approach, WPQ entries are augmented with a volatile (i.e., no need for extra ADR support) bitmask to indicate which partial parts of a metadata block are in this block. Each time there is a partial update, we check if the security metadata block exists in the WPQ, and if so, we update the bitmask and merge the partial update within its original metadata block in the WPQ. Upon an eviction of a WPQ entry, we check the bitmask, if all bits are set to 1 (e.g., ciphertext block or metadata block has all its fields updated while in WPQ), then we persist the block in its original location. Otherwise, based on the number of partial updates within a metadata block derived from the bitmask, we can choose to place it in PCB or simply persist the full block.

Fortunately, we found that an augmented version of PCB-before-WPQ, where we check the addresses of partial updates in the PCB upon each partial update such that they are merged, can minimize the pressure on the WPQ and obtain similar performance as in PCB-after-WPQ. Thus, throughout our evaluation we use such an augmented PCB-before-WPQ with 8 entries of the WPQ devoted for PCB while the remaining 56 entries are used as WPQ, compared to a 64-entry WPQ in the baseline. Figure 5.6 depicts our adopted approach.

5.2.4 Recovery Scheme

Upon a crash, the PUB will contain more recent values than the values in the security metadata blocks in their original locations in NVM. Thus, upon a restoration from crash, these security metadata need to be merged with their original locations in NVM (rather than the PUB buffer in NVM). It is critical to note that even though the most recent counters and MACs are not persisted in-place, they can still be verified using the typical mechanism relying on the integrity tree. Thoth relies on prior works to ensure crash consistency of the integrity tree, e.g., Anubis(65), but reduces the number of writes needed to be able to reconstruct the verifiable integrity tree. For instance, in Anubis(65), the counters and MACs must be recovered, however they are verified through a persistent up-to-date integrity root. We leverage the same mechanism, however, before we reconstruct the then-to-be-verified tree, we need to recover the counters and MACs; such recovery of MAC/CTRs was previously done by leveraging ECC bits or co-location, which is no longer feasible in emerging memory interfaces. Thus, **Thoth's responsibility is to merely merge the updates in PUB with the tree (and MAC blocks) to be re-constructed before verification**. To do that, the first step is to scan through the partial updates in PUB in a reverse order (i.e., oldest entry to youngest entry), read it, then read the corresponding metadata blocks, merge the updates (counter and MAC) in their corresponding blocks and write them to memory. To recover the MAC, we will fetch the corresponding ciphertext, compute two levels of MAC, and use the second level of MAC to verify the fetched ciphertext. Note that the potential replay attack will be detected later when the integrity of counters is verified using typical mechanism relying on the integrity tree. Once all PUB entries are incorporated, the tree reconstruction can be done as suggested in Anubis(65). Note that to speed-up recovery time, Anubis already records the addresses of the blocks lost from the cache in a shadow region. Thus, Thoth would first recover the entries in the WPQ, then leverage Anubis' fast recovery mechanism by reading its shadow address tracking and start the tree reconstruction of the inconsistent parts, all before the tree verification starts.

Thus, in addition to the sub-seconds of recovery time for Anubis(65), we add a marginal extra recovery time of 7 seconds even for a PUB as large as 64MB[4]. We believe that 7 seconds of recovery time is marginal compared to other boot-up and OS aspects upon system startup.

5.3 Evaluation

5.3.1 Simulation Setup

We use GEM5(17), a cycle-level simulator to evaluate the performance overheads of Thoth. As illustrated in Table 5.1, we simulate 4 X86-64 Out-of-Order cores with 32GB DDR-based PCM. We also use 4 database benchmarks from WHISPER(44) and one in-house benchmark (Random Array Swap). For each benchmark, we fast-forward to a point where the application has run at least 5000 transactions on each core and also insert partial update entries into the PUB using realistic secure metadata generated during the fast-forwarding phase. In the evaluation, we start evicting entries from the PUB when it is 80% full and this threshold is met after the fast-forwarding phase. We set the off-chip PUB size to be 64MB, which incur less than 1% storage overhead with off-chip memory capacity of 32GB. We incorporate a counter cache, MAC cache and MT cache with default size of 64 kB, 128 kB and 256 kB, respectively, in the memory controller. We model a 10-level MT over NVM with lazy update and a 4-level MT over the secure metadata cache with eager update. In our model, a single MAC computation takes 40 cycles, similar to prior work(40; 12; 65). Table 5.1 lists the detailed configuration for our simulation. For all applications, we use 128B as the default transaction size. The WHISPER workloads are command-line configurable for multiple transaction sizes, and we implement our in-house benchmark with similar functionality by setting the swapped array length to the transaction size.
Baseline machine setup: The baseline machine adopts strict persistence for counters and MACs and allows persisting MT nodes through natural evictions, since these can be verified using an eagerly updated root(65). Since each counter is persisted directly to NVM, besides updating the 4-level small merkle tree over the secure metadata cache, we calculate another hash for the last level of the merkle tree. We set the WPQ to start draining when it is 50% full so that secure metadata from the same cache block that arrive in a short time period can be coalesced.

[4]This can be calculated by latency needed to each of the PUB blocks, reading their corresponding MAC, ciphertext and counter blocks, computing 2 leves of MAC, then updating the counter and MAC blocks of each PUB blocks' entries.

Table 5.1: Simulation Configuration Parameters

Processor	
Core	4 Cores, X86, OoO, 4GHz
L1 Cache	2 cycles, 64KB,2-Way
L2 Cache	20 Cycles, 2MB, 8-Way
LLC	32 Cycles, 16MB, 16-Way
WPQ size	64 entries in baseline; 56 entries in Thoth
DDR based PCM Memory	
Size	32 GB
Access Latency	read latency 150ns. write latency 500ns.
Secure Memory Parameters	
AES Latency	40 Cycles
Single Hash Latency	40 Cycles
Integrity Tree	a 10-level 8-ary Merkle Tree over NVM; a 4-level 8-ary Merkle Tree over secure cache
Tree Update Policy	Lazy Update for MT over NVM Eager Update for MT over secure metadata cache
Partial Update Buffer	WTSC, size = 64MB
Persistent Combining Buffer	8 entries

Number of partial updates / cache block	9 updates in 128B block;
	19 updates in 256B block;

Counter cache size	64kB; 4 way
MAC cache size	128kB; 8 way
Merkle Tree cache size	256kB; 8 way

5.3.2 Overall Performance

Figure 5.7 compares the overall speedup achieved by Thoth over the baseline for both 128B and 256B cache block sizes. Each workload is configured with a transaction size of 128B. Thoth achieves similar speedup using WTSC scheme and WTBC scheme. Thoth achieves an averaged speedup of 1.22× and 1.16× for cache block sizes of 128B and 256B, respectively. The performance advantages of Thoth stem largely from the reduction in write traffic compared to the baseline, as shown in Figure 5.8. Thoth reduces the number of writes by an average of 32% and 37% for block size of 128B and 256B, respectively, over the baseline system. The 256B blocks are able to pack more partial updates per block and coalesce more entries, leading to a greater reduction in writes. The swap benchmark shows

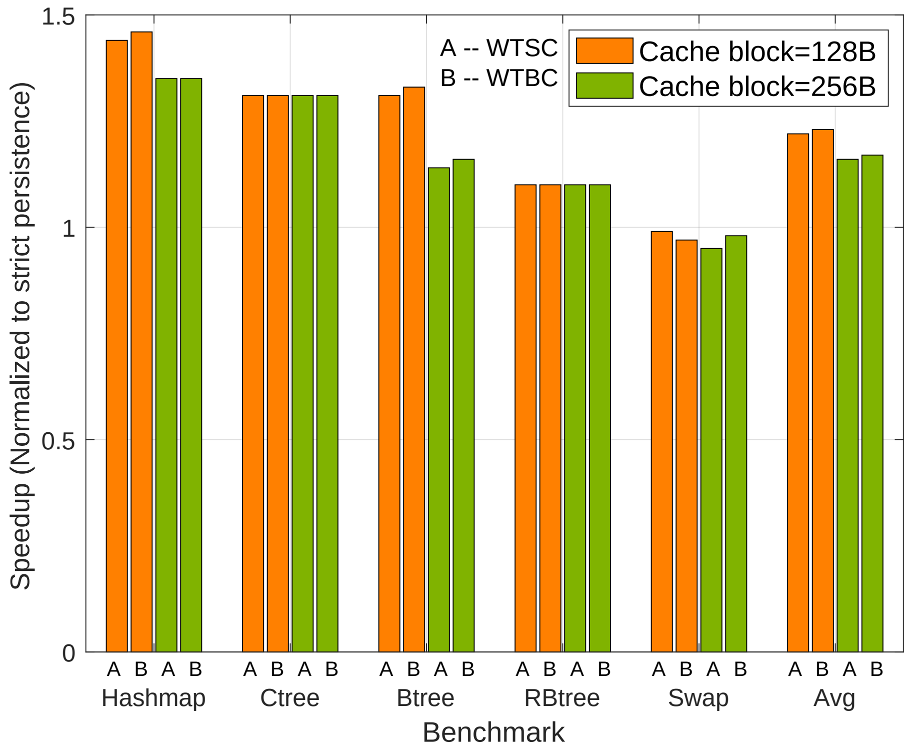


Figure 5.7: Speedup of Thoth with WTSC and WTBC scheme (transaction size = 128B)

the opposite behavior(i.e., more reduction of writes in 128B cache block) because more evictions of secure metadata in Thoth with 256B cache blocks. Even though that higher reduction in writes for 256B cache blocks, the speedups are less for 256B cache block. This is because writes in baseline are reduced by 256B cache block. Therefore, the performance of our baseline is improved by using 256B cache block, which leads to less speedup in Thoth using 256B cache block.

Among the workloads, the swap benchmark does not achieve any speedup when using Thoth and even degrades in performance a little. This is partly an effect of the relatively small transaction size of 128B. Because swap merely exchanges two arrays that are allocated contiguously in memory, it touches few memory locations and induces relatively few secure metadata writes to memory as compared with the other workloads. While Thoth successfully eliminated 20% and 15% of swap's writes to memory, for 128B and 256B respectively, this did not yield a speed-up.

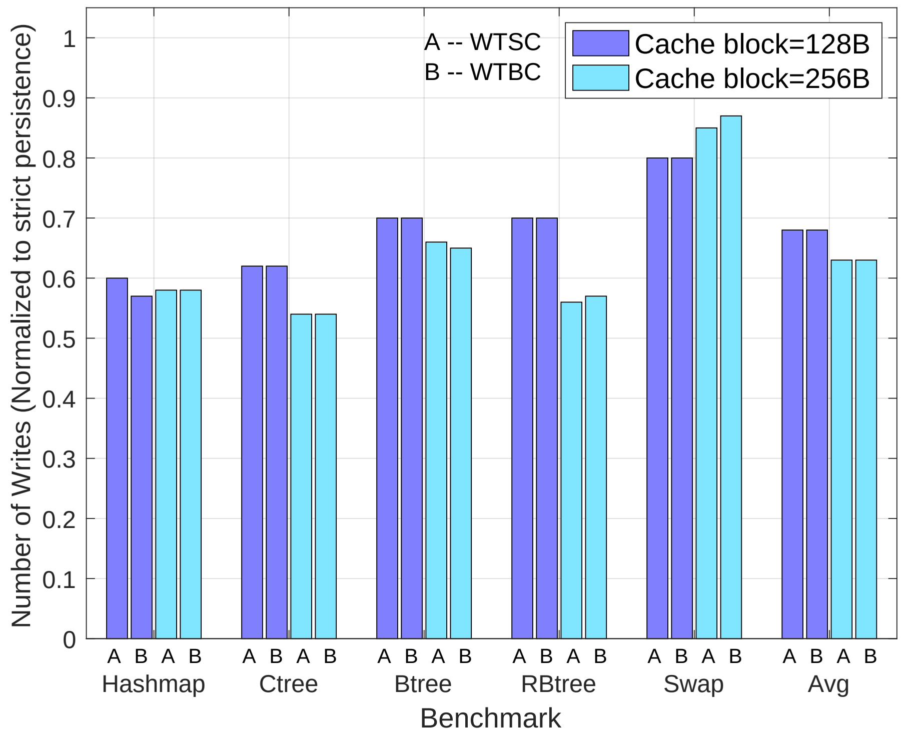


Figure 5.8: Number of Writes of Thoth with WTSC and WTBC scheme (128B transactions)

5.3.3 Sensitivity to Different Transaction Size

We also study the performance of Thoth over multiple transaction sizes. We run each application with transaction sizes of 128B, 512B, 1024B and 2048B using cache block sizes of 128B and 256B. The average speedup of Thoth with 128B blocks is 1.22×, 1.23×, 1.19× and 1.19× when transaction size is 128B, 512B, 1024B and 2048B respectively, as shown in Figure 5.9. The average speedup of Thoth with 256B blocks is 1.16×, 1.17×, 1.14× and 1.19× when transaction size is 128B, 512B, 1024B and 2048B respectively, as shown in Figure 5.9. In some of the benchmarks, Thoth achieves less speed up over the baseline as transaction size increases because the baseline behavior improves. Larger transactions create more opportunity for coalescing secure metadata writes in the WPQ in the baseline machine, reducing the gains in Thoth.

To help explain this trend, Table 5.2 shows the average percentage of writes to ciphertext for both the baseline and Thoth. The percentage of writes for ciphertext in Thoth mainly depends on the eviction rate from the secure metadata cache and the coalescing effect in the PCB. As the transaction size increases, the relative benefit of coalescing in the PCB decreases with respect to coalescing in the WPQ in the baseline, as shown in Table 5.3. This is

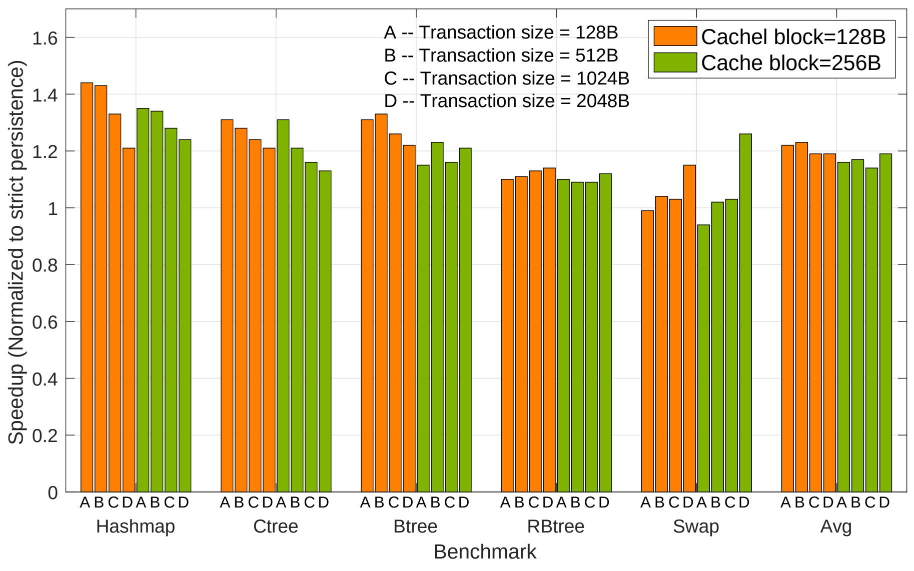


Figure 5.9: Speedup of Thoth for transaction sizes of 128B, 512B, 1024B, 2048B

Table 5.2: Averaged percentage of writes for ciphertext on transaction size of 128B, 512B, 1024B, 2048B

Type(Cache block)	**Percentage of merged partial updates**			
	128B	512B	1024B	2048B
Baseline(Cache block=128B)	45.52%	49.28%	52.68%	58.15%
Baseline(Cache block=256B)	41.35%	44.15%	47.24%	51.30%
Thoth(Cache block=128B)	68.35%	67.97%	68.78%	73.59%
Thoth(Cache block=256B)	67.09%	67.00%	69.60%	76.24%

because the on-chip PCB can only coalesce consecutive updates for the same partial update (same minor counter/MAC). With larger transaction sizes, these consecutive updates are less likely to reside in the PCB at the same time. However, the WPQ can coalesce any two updates to the same secure metadata cache block, giving it more flexibility to coalesce writes. Also, the WPQ is larger than the PCB, giving it another advantage of more entries over which it can attempt to coalesce. In spite of these advantages in the baseline, Thoth still achieves a significant speed up across all configurations. Thoth reduces the number of writes on an average by 32%, 28%, 24% and 20% for 128B cache block and 37%, 33%, 31% and 31% for 256B cache block. The trend shows Troth reducing writes by a smaller percentage as the transactions grow larger because of better coalescing in the WPQ of the baseline machine as transaction size increases.

Btree of 256B cache block, rbtree and swap have a different trend. They achieve more

Table 5.3: Averaged percentage of partial updates merged in PCB on transaction size of 128B, 512B, 1024B, 2048B

Cache block	**Percentage of merged partial updates**			
	128B	512B	1024B	2048B
Cache block = 128B	74.36%	57.68%	44.26%	34.25%
Cache block = 256B	87.88%	80.51%	71.17%	62.74%

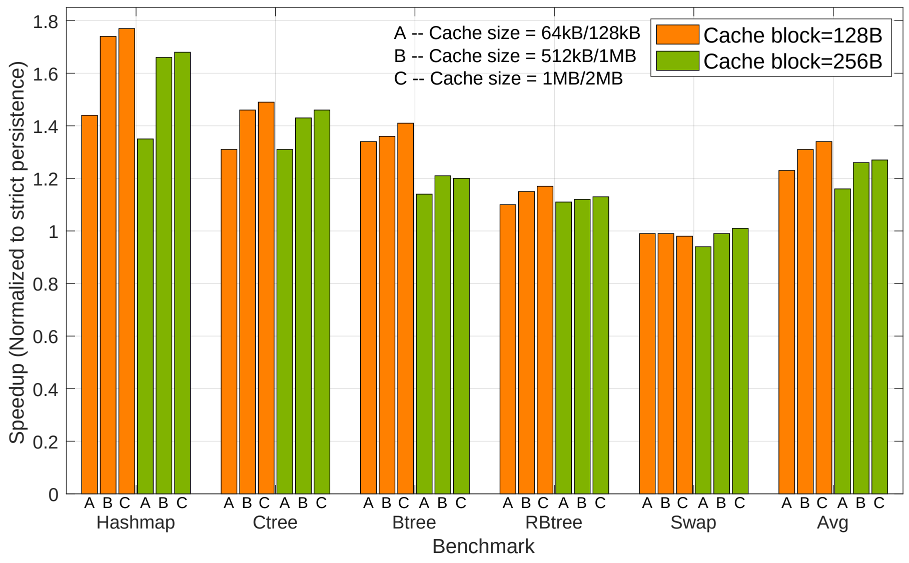


Figure 5.10: Thoth's Speedup for various counter/MAC cache sizes.

speedup as transaction size increases due to the number of writes in the baseline and/or more reduction in writes caused by less evictions from metadata cache with larger transaction size. In some cases(e.g., Swap) where performance increases with transaction size, the number of memory writes with larger transaction size in the baseline is significantly higher than the one in smaller transaction size, which signify the impact of memory writes reduction on performance.

5.3.4 Sensitivity to Secure Metadata Cache Size

We further characterize Thoth in the context of larger secure metadata caches. We evaluate the performance of Thoth when varying the counter/MAC cache size to 64kB/128kB, 512kB/1MB and 1MB/2MB. The average speedup of Thoth with a 128B cache block is 1.22× with the smallest secure metadata cache sizes to 1.34× at the largest sizes as shown in Figure 5.10. Similarly, the average speedup with a 256B cache block is 1.16× for the smallest cache sizes up to 1.28× for the largest. These trends show that Thoth achieves even more speedup with larger metadata cache sizes. This is because Thoth allows secure metadata to be persisted through natural eviction from cache, and with larger metadata cache sizes, there will be fewer evictions and, hence, fewer write backs as well.

5.4 Related Works

Secure NVM: In addition to enabling crash-consistency, various secure NVM works optimize for different things. For instance, Anubis(65), enables fast and ultra-low recovery time of integrity trees and counters. Meanwhile, Osiris(60) allows recovery of encryption counters regardless of recovery time. Soteria(66) hardens integrity trees to improve reliability. Dolos(27) minimize the latency for persistent transactions through a decoupled security unit. Janus(40) enables efficient scheduling of security memory backend operations. All these works need a new vehicle to persist their security metadata when no ECC or extra metadata bits are available for the processor, and hence Thoth can help realizing them in future memory interfaces.

Overloading ECC Bits: Most of the state-of-the-art solutions in secure NVM (60; 65; 12; 19; 58; 27; 66; 64; 59; 68) builds on the idea of co-locating such security metadata with data to minimize the write amplification for implementing crash-consistent secure NVMs. Other works, focus on selectively choosing which data to persist(39). Thoth provides a new vehicle for prior works through leveraging off-chip partial update buffers in the absence of processor-controlled ECC bits or extra pins, as the case in future memory interfaces. For demonstration purposes, we presented Thoth enabling a state-of-the-art secure NVM work(65) in future interfaces. Some other works(68) leveraged WPQ-like structures to hold metadata blocks before they are persisted, to enable merging more writes before writing to NVM; this is already captured in our baseline.

5.5 Summary

In this chapter, we propose Thoth, a novel off-chip persistent buffer that can enable crash consistency in future memory interfaces with minimal write amplification. Thoth works by combining several partial secure metadata updates into one memory block write and persists them in a large off-chip persistent buffer in NVM. Based on our evaluation, Thoth improves the performance by an average of 1.22× (up to 1.44×) while reducing write traffic by an average of 32% (up to 40%) compared to the baseline Anubis when adapted to future interfaces.

CHAPTER

6

ATHENA: TRANSPARENT SOFTWARE-BASED DATA CONFIDENTIALITY FOR EMBEDDED SYSTEMS

IoT devices are being deployed at an unprecedented scale which mandates measures for protecting data confidentiality, however there currently exists no solution between both extremes: hardware-only memory solution and no memory encryption at all. Unfortunately, hardware-only solution is not applicable for already deployed IoT devices without a hardware solution, and also not suitable for systems with low cost/power/area budget. Meanwhile, having no encryption at all is not an option given the increased attack surface and the type of data can be processed by IoT devices. Thus, we devise compiler-based software memory encryption framework that fills such a gap and allows systems without (or those cannot afford) hardware-based memory encryption to still protect data confidentiality.

A software-based solution can be implemented by programmers, however this entails significant manual efforts and error-prone; it requires a programmer who is familiar with the hardware architecture, understands the threat model, and fully aware of the impact all code transformation steps (e.g., compiler) on the program. Meanwhile, a programmer-independent solution is likely to introduce significant performance overheads. For instance, a compiler-based solution that naively transforms every load/store operation to complete decryption/encryption will introduce significant performance overheads due to the following reasons. First, the data locality is not exploited as the data sill needs be encrypted/decrypted on every access, due to the limited visibility. Even if the data is cached within the trusted domain, a naive scheme would still need to assume it came from memory; similarly, even if for store operation, the data is assumed to go all the way to the memory due to the absence of a security unit at the memory controller. Encountering encryption/decryption on each access leads to significant overheads. Second, the function calls needed on each access, i.e., transformation overhead, significantly impacts execution time. Our evaluation shows that such a naive solution could lead to an ***average system slowdown of 45.87x (up to 148.85x) relative to a non-secure system***. Meanwhile, a solution that allows exploiting data locality will require careful management of such data to ensure it cannot escape the trusted domain (i.e., chip boundaries). Moreover, an optimal solution would minimize the transformation overhead by eliminating the need for a functions call for each access.

To address the above mentioned challenges, we present ***ATHENA***, a compiler-based framework that exploits data locality and minimizes transformation overhead, to provide a practical memory security solution for embedded systems. ATHENA aims to (i) identify safe internal hardware structures that can be leveraged to minimize the encryption/decryption overheads via exploiting data locality; (ii) automatic secure placement and eviction of data without programmer intervention; (iii) optimize and reduce code transformation overheads using compiler-directed memoization; and, (iv) leverage the available hardware Memory Protection Unit (MPU) in novel ways to reduce overheads. We evaluate ATHENA on five machine learning workloads designed for embedded system that are cost/power/area constrained. All applications are first compiled using GCC compiler with our support for secure memory and run on STM32F769I-Eval board (a representative evaluation board for embedded systems). Experiment results show that ATHENA reduces execution time to an average of 0.13×, compared to a baseline compiler-based design.

6.1 Motivation & Preliminary Results

6.1.1 Design Scope

Our work aims at low-end embedded devices without hardware memory encryption module. The targeted MCUs have internal FLASH memory and internal RAM. A Memory Management Unit (MMU) does not exist to support virtual memory, but they often include a Memory Protection Unit (MPU) which can provide access control to memory and be used to improve the performance. Given the limited resources of the target domain, we also assume the system runs a single-threaded bare-metal application.

6.1.2 Challenges

To better understand the challenges for an automated software-only solution, we start with a baseline that achieves two main objectives: (1) it requires minimal programmer intervention, and (2) it ensures that all of the workload's data are encrypted when stored off-chip.

To meet the first requirement, we use a compiler to automatically modify the application to perform encryption and decryption on all accesses to data in memory. This saves a substantial amount of programmer effort in rewriting code. Even though a programmer may know algorithmic details and have some idea about data locality that would help in reducing encryption and decryption overheads, the programmer would also take on the responsibility of ensuring that no data is accidentally omitted or written to off-chip memory unencrypted. Hence, using a compiler ensures low programmer effort and uniform treatment of all data in the program.

To meet the second requirement, we use the compiler to insert the necessary memory encryption and decryption operations at every memory operation in the program. Instead of executing a load to off-chip memory, the load is transformed into a decryption operation that returns the requested decrypted data. Likewise, all stores are transformed to an encryption operation that stores encrypted data. The transformed code ensures that all intermediate plaintext state is held on-chip in trusted memory locations and cannot be evicted to off-chip memory. The combined effect is that all data outside the TCB is encrypted.

Encrypting and decrypting data at each store and load, respectively, is conservatively correct but comes with overhead. Ideally, we should minimize redundant encryption and decryption operations. As part of the baseline design, we use the compiler to aggressively

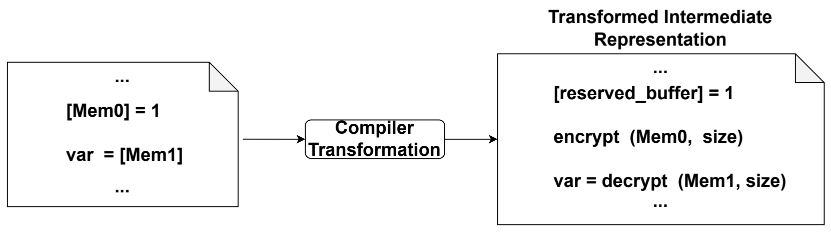


Figure 6.1: The Baseline Design

optimize the code, using O2 optimizations, before applying our transformation. While this will eliminate many provably redundant memory operations, some will remain. In general, it is challenging to know if the data being accessed has been encrypted or decrypted already due to control flow, function calls, and pointer aliasing. Furthermore, it also would be critical to know if the data had been modified since that last operation, potentially invalidating it. We do not consider any further optimizations in the baseline, but we do consider them as part of our optimized design in Section 6.2.

Finally, it is worth noting that not all loads and stores are destined for off-chip memory. For example, we assume the stack is held on-chip. Hence, the operations inserted by the compiler contain additional checks performed at runtime to ascertain whether or not security operations are required, and if not, the load or store is carried out without encryption.

Compiler Transformation and Runtime Actions: We briefly describe the baseline compiler transformation. As shown in Figure 6.1, two functions (*encrypt() and decrypt()*) are provided in the baseline design, and all memory load/store statements are replaced by *encrypt()*/*decrypt()* respectively. The functions are implemented in a library that is linked with each transformed program.

The encrypt and decrypt functions will check the memory address and take action according to the address range. If the address is within the on-chip memory range, operations proceed normally and directly access to on-chip memory, otherwise, a security operation is required before access. To securely encrypt/decrypt data, the working memory of the encrypt and decrypt functions must be kept in on-chip memory to ensure that no data is evicted to off-chip memory inadvertently. Therefore, a block of on-chip memory is reserved for the input and the generated output of the cryptographic process. The size of the reserved on-chip memory depends on the encryption granularity. For example, counter-mode memory encryption using AES typically assumes an encryption block size of 16 bytes. Hence, 16 byte of on-chip memory is reserved for each of these: the IV, the generated OTP, and the plaintext. We only reserve enough memory to support a single security operation at a time. For both the read and write in the baseline, the values in these buffers are discarded and overwritten by each new operation.

For the read operation, the aligned 16 bytes of data is fetched from off-chip memory and decrypted. After decryption, the requested word is returned to the program. The write operation is more complex. First, the aligned 16 bytes of data is fetched and decrypted using the old counter value. Then, the plaintext is updated with the newly written data. Finally, it is re-encrypted using the incremented counter value and written back to the off-chip memory block.

Overhead of Baseline: We test the performance of the baseline design over 5 Machine Learning benchmarks (45; 16) as listed in Table 6.1. The model of each benchmark is stored and encrypted in the off-chip memory. For each benchmark, we test the execution time of a single inference. We compare the execution time of the baseline design with the non-secure design. The extra execution time in the baseline design is mainly attributed to: (1) memory encryption/decryption operation, (2) data movement, (3) memory address range check, and (4) our transformation. Figure 6.2 shows and breaks down the performance overhead of the baseline design over the non-secure design. In total, the baseline design incurs an average performance overhead of 46×. By removing overhead of (1), the average performance overhead is reduced to 7.57×. By removing overhead of (1) and (2), the average overhead of execution time is reduced to 5.54×. By removing overhead of (1), (2) and (3) (i.e., only considering overhead introduced by our transformation), the average overhead is reduced to 2.74×. The significant performance gap between the design with and without decryption and encryption indicates a large potential performance gain by reducing frequent memory encryption operations. Therefore, we need not only a tool to automatically modify memory statements in applications but also a tool with the capability of reducing frequent memory encryption operations by leveraging data locality, which may also be beneficial for reducing overhead of (2). However, leveraging data locality may introduce new overhead, i.e., checking whether data is already decrypted & cached on chip, address mapping/translation between on-chip & off-chip memory. Therefore, we need a tool that can leverage data locality to reduce memory encryption/decryption overhead and meanwhile reduce other aspects of overhead by optimized compiler transformation.

Table 6.1: Benchmark Description

Benchmark Name	Model Size
Fast Artificial Neural Network (FANN)	274 KB
Anomaly Detection (AD)	270 KB
Person Detection (PD)	325 KB
Image Classification (IC)	96 KB
Keyword Spotting (KS)	52.5 KB

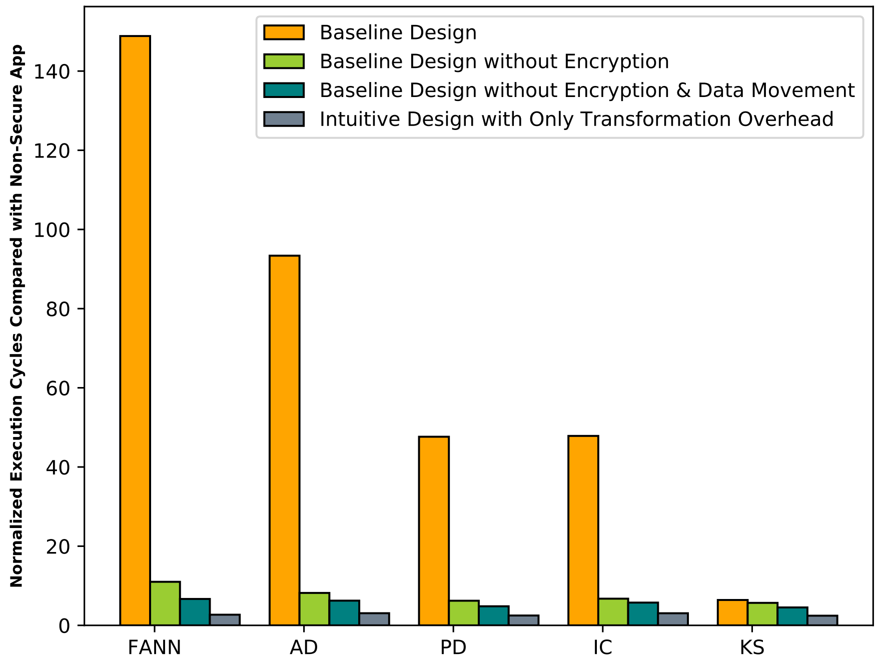


Figure 6.2: Performance Overhead of The Baseline Design over the Non-Secure Design

6.2 Design of ATHENA

6.2.1 Design Objectives & Principles

The main design objectives for ATHENA emphasizes three aspects: confidentiality, transparency and high performance.

Confidentiality: ATHENA provides application-transparent confidentiality for application data stored in the off-chip memory. Accordingly, the first objective of ATHENA is to guarantee application data confidentiality. The first design choice is determining where to place application data among the on-chip and off-chip memories. Since the on-chip memories are usually much smaller, it is generally infeasible to place the entire application code and data in on-chip memory. For any data placed off-chip, ATHENA needs to provide a software layer of encryption/decryption process between the application and the off-chip memory. ATHENA achieves this through a Secure Library that mediates access through encryption and decryption operations.

Transparency: In ATHENA, the application needs to be oblivious to extra actions due to encrypting/decrypting off-chip memory access. Depending on how the application's data and code are placed into memory, transparency can be easy or difficult to achieve using a compiler. For example, if application code is fetched and executed from off-chip memory, there is no suitable way for a compiler to insert the necessary encrypt and decrypt operations. Likewise, were the runtime stack to be stored off-chip, then spills and fills to the stack that are commonly inserted during the final stages of compilation (e.g. register allocation) would need to be captured and secured before writing to memory. From a design perspective, it is quite challenging to insert transformation late in the compile process after register allocation, and few, if any, compiler infrastructures support this robustly. Hence, it is critical to place the instructions and stack in trusted memory. Fortunately, these tend to be small regions of memory and can fit within typical on-chip memory capacities(38). Stack size of our benchmarks is within 1KB and code size is smaller than 800KB. The remaining global and heap data can be placed either in trusted or off-chip memory. Accesses to global or heap data would arise through loads and stores in the program, and the compiler can easily identify them and automatically modify them to support the needed confidentiality operations.

Performance: Another design objective of ATHENA is how to reduce performance impact of adding memory encryption. Unlike in hardware solutions for secure memory, the overhead of memory encryption now will occur on the CPU and appear on the critical path of loads and stores. The high cost of these operations are seen in the baseline's execution overhead of 46×. Several factors lead to this slowdown.

Granularity. There is a granularity mismatch between loads and stores in the program and the granularity of encrypted data. Data is encrypted in 16-byte blocks but we access it through loads and stores that only want 1 or a few bytes per access. Much of the work goes to waste because the block is discarded by the next operation.

Lost Locality. There are frequent encrypt and decrypt operations to the same 16-byte blocks of encrypted memory that occur due to frequent loads and stores to the same 16-byte block. In hardware-based solutions for secure memory, plaintext would be held in the cache and reused many times rather than discarding the plaintext data between operations. ATHENA must exploit a similar effect.

To solve the granularity and locality problems, ATHENA adopts a Secure Mapping

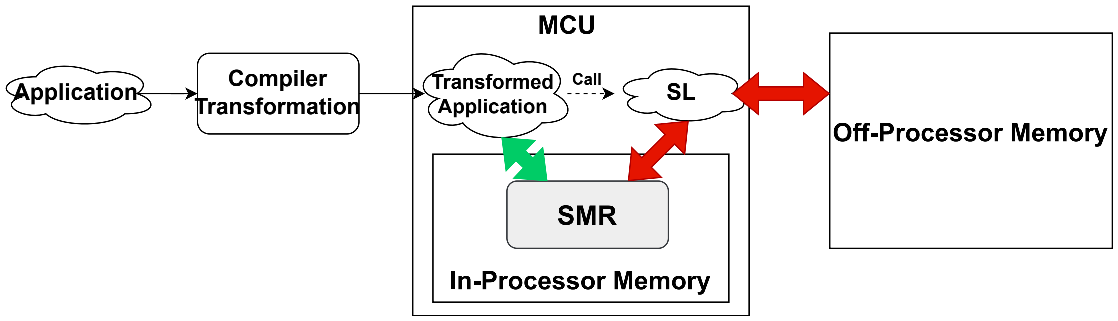


Figure 6.3: High Overview of ATHENA

Region (SMR) in trusted memory that acts as a software cache to hold the plaintext for recently decrypted blocks. We will describe how the SMR can be used to serve loads and stores and eliminate many security operations.

Code Transformation Overhead. The replacement of loads and stores with more complex decrypt and encrypt security operations adds overhead. Even though we aggressively apply O2 optimizations in advance of the transformation, the overhead is 2.74× on average. ATHENA adopts two strategies to reduce this overhead: (1) using the compiler to customize transformation depending on the context of the memory operation (ATHENA Memoization of Translation in Section 6.2.5), and (2) using the hardware MPU and trap handler to reduce transformation overhead (ATHENA MPU Silent Translation in Section 6.2.5).

6.2.2 Overview of ATHENA

In this section, we present an overview of ATHENA's design, as shown in Figure 6.3. ATHENA consists of a Secure Memory Pass implemented in GCC and a runtime library called the Secure Library (SL). As in the baseline design, the compiler pass replaces all loads and stores in the application with a set of instructions that secure the access. The exact approach for this transformation is different from the baseline and will be described after explaining some of the working details of the SL.

All accesses to off-chip memory must go through the SL. This makes it possible for the SL to optimize for performance and locality. The most important optimization is that the SL maintains a Secure Mapping Region (SMR) in on-chip memory that serves as a software cache for recently decrypted blocks. The SL securely moves data into and out of the SMR on demand as data is accessed at a large granularity (e.g. 16 bytes) typical for memory encryption. After data is placed in the SMR, an application can securely update or read plaintext inside the SMR. The SMR is critical for achieving higher performance by allowing many reads or writes to benefit from a single round-trip encrypt and decrypt operation to off-chip memory.

When a load or store targets an address of which its data is already in the SMR, no overhead is incurred for encryption or decryption. Instead, the off-chip address is translated to its temporary address in the SMR and then the access proceeds. The compiler pass supports this by inserting an address translation on the address operand of each load or store.

The Secure Library and Secure Memory Pass are the core of ATHENA and described more in the next two sections. Then, ATHENA is improved further by reducing translation overheads (Section 6.2.5) and optimizing the encryption algorithm (Section 6.2.6).

6.2.3 Secure Memory Pass

ATHENA's Secure Memory Pass modifies all loads and stores by translating their original address into a secured one within the SMR. The Secure Memory Pass is implemented as part of the GCC compiler using the Gimple representation. At the Gimple level, memory operations are represented as an assign statement, much like a pointer dereference in C. For a store, the left-hand-side (lhs) is a memory reference to some address; the right-hand-side (rhs) is the value operand. Loads are represented with the memory operation on the right-hand-side. The Secure Memory Pass visits all assign statements that are loading or storing to memory and transforms them.

Figure 6.4 shows an example of transforming a store statement. First, function regular_secure_store is inserted before the memory statement to translate the memory address to a SMR address. regular_secure_store is a function provided in run-time SL. regular_secure_store takes one argument that specifies the original memory address and attempts to find a mapped address for the external-memory block inside SMR. If successful, it returns the translated SMR address. As a result, the memory statement will access the translated SMR address. ATHENA conservatively transforms every memory statement that may access off-chip memory, and hence some translations are unnecessary because the address is within on-chip memory (e.g. stack or globals mapped on-chip). If the memory statement accesses the on-chip memory, the function simply returns the passed argument as the address.

Loads are handled in a similar way. The load memory statements are transformed to call the regular_secure_load provided in the SL. Figure 6.5 illustrates the procedure regular_secure_load function. In Figure 6.5, The to-be-translated address is passed into

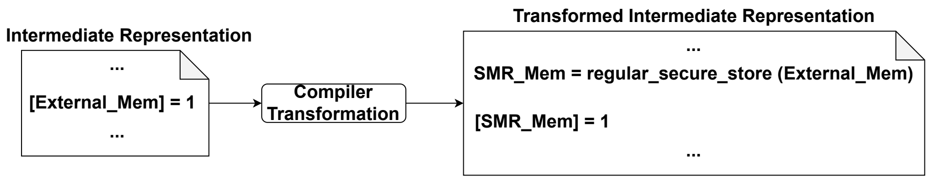

Intermediate Representation
...
[External_Mem] = 1
...
Compiler Transformation
Transformed Intermediate Representation
...
SMR_Mem = regular_secure_store (External_Mem)
[SMR_Mem] = 1
...

Figure 6.4: Example of ATHENA Transformation of Memory Access

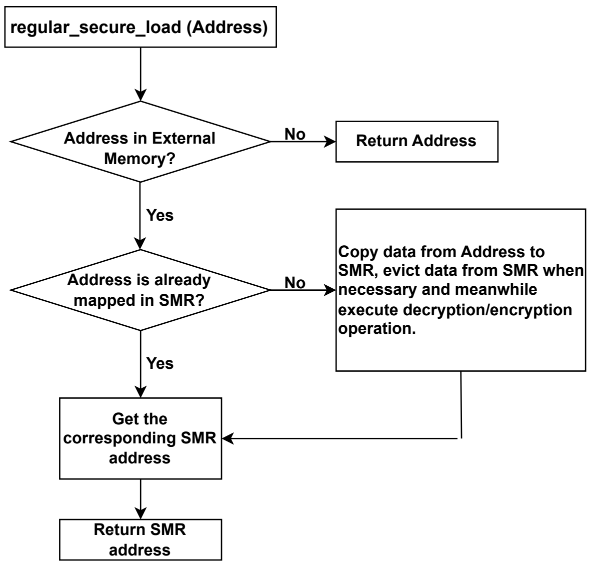

regular_secure_load (Address)
Address in External Memory?
No
Return Address
Yes
Address is already mapped in SMR?
No
Copy data from Address to SMR, evict data from SMR when necessary and meanwhile execute decryption/encryption operation.
Yes
Get the corresponding SMR address
Return SMR address

Figure 6.5: Procedure inside regular_secure_load function

regular_secure_load. First, SL determines whether the address requires a translation. If the address is in the internal memory or unprotected external memory, regular_secure_-load returns with the original address immediately. Otherwise, SL continues to check if the address is already mapped into SMR. If the address is already mapped, SL gets the corresponding SMR address and returns with it. Otherwise, SL needs to first securely copy data from external address to SMR and execute necessary memory decryption operation meanwhile. Then get the SMR address and return with it. Function regular_secure_store has similar process except that dirty bit of the written SMR block need to be set inside function regular_secure_store.

Ensuring Correctness of Translation

The compiler must correctly and transparently translate off-chip addresses into the SMR. To ensure this is done correctly, we place constraints on what the compiler can do. (i) The translated address may not be used outside its intended load or store. This is critical for maintaining transparency and correctness. First, for correctness, we cannot assume that any other load or store with the same address operand will translate to the same SMR entry because SMR entries may be evicted by intervening memory accesses. However, we reconsider this later in Section 6.2.5 when we work to reduce translation overheads, and we add support to prevent block evictions from the SMR.

(ii) For correctness and transparency, it is important that the SMR address is not passed to a function or stored into application data structures. If passed to a function, it could be stored to memory or incorporated into program state, and would from that point onward appear to be a trusted memory address. However, this would make the translated address within the SMR part of the program state leading to a loss of transparency and other undesirable behaviors like illegal accesses to the SMR and various errors and bugs.

(iii) When translating into the SMR, the alignment and size of the load or store must result in an access fully contained within a single SMR block. ATHENA transforms potential unaligned memory statements to force them invoke special service provided in Secure Library in which the access boundary will be checked and corresponding actions will be taken if the access is across two SMR blocks.

We use MPU as a helper to check all illegal accesses issued from applications to off-chip memory. At the beginning of execution, the MPU is programmed to make the off-chip memory region with application data non-readable and non-writable. The SL changes the access policy to be readable or writable as needed and resets the access policy before leaving the SL and returning to program code.

6.2.4 Run-time Secure Library

Run-time Secure Library is responsible for the following: 1) management of the SMR, 2) performing the memory encryption/decryption operations, and 3) address translation from off-chip memory locations to the SMR.

Management of the SMR

The Secure Mapping Region (SMR) is managed as a cache. Memory reads and writes issued to off-chip memory may cause allocations of a memory block in the SMR, which we call a SMR block. ATHENA supports flexible configuration of SMR, such as size of SMR, size of a SMR block and set associativity of the SMR. While a variety of associativities are possible, the SMR is configured as fully associative to prevent conflict misses and avoid additional encryption operations caused by conflicts.

Secure Memory Operations

Memory encryption and decryption operations are invoked as blocks are allocated and evicted from the SMR. Upon allocation of a secure SMR block, decryption is applied on the newly allocated block. Upon eviction, encryption is applied and it is written back to off-chip memory.

ATHENA supports flexible memory encryption because it's fully implemented in software, hence a variety of algorithms could be deployed for greater performance or increased security. For now, ATHENA implements counter-based encryption with AES algorithm. Secure metadata (counters) are stored in the off-chip memory given its larger size. Each SMR block is associated with a base counter value. The size of the SMR block needs to be aligned with the encryption block size. If a SMR block contains multiple sub-blocks that are the same size as the encryption block size. Each sub-block is encrypted by incrementing the base counter value as part of the Initialization Vector (see Section 2.1).

Address Translation with TLII

A mapping table is maintained in the SMR to record the mapping between an SMR block and an off-chip memory block. For each location in the SMR, a mapping table entry records the off-chip block and the SMR block offset. Due to the fully associative nature of the SMR, finding a matching entry would naively require a search of the entire mapping table.

To reduce the overhead of translation, ATHENA incorporates a fast lookup technique called **Two-Level Indirect Indication (TLII)**. The idea is that we can store an *indirect indication* (a prediction) of the mapping of where the block is mapped and then validate it rather than search the mapping table every time. This allows us to have a bigger fully-associative SMR. We developed two such techniques that work together for higher performance and security.

The first approach uses the first several bits of the external block in off-chip memory to store the indication. When a block is allocated in SMR, the copy off-chip is temporarily unneeded. ATHENA repurposes the first several bits to hold the offset of the SMR block. When we need to translate an address, the first bits of the block are accessed to obtain its possible SMR offset. The guessed SMR offset is used to determine whether that block is in the SMR. If it's a hit, we can use that SMR entry. Any block currently in the SMR should hit. Blocks that are not yet mapped will miss, but given the high overhead of decryption, this imposes only a small additional overhead to perform this check first. For cases where we find the mapping, the savings are significant because one access to off-chip memory is

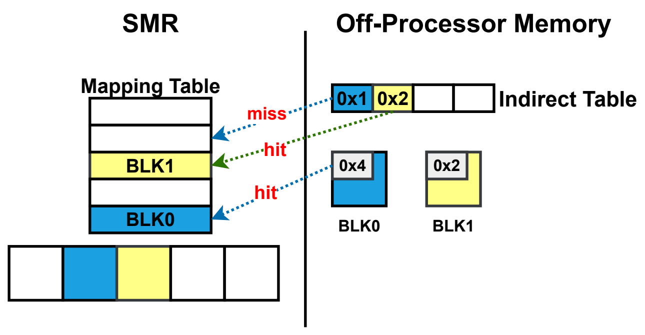


Figure 6.6: Illustration of Two-Level Indirect Indication (TLII)

much quicker than searching a large table.

However, one downside of this approach is that for every translation it requires changing the MPU access policy to allow access to the first bits of an off-chip memory block, which can take 40 cycles on average to enable the access and disable it before returning to the application. To avoid frequently changing the MPU access policy and save cycles of changing access policy, we incorporate an additional level of indirect indication in the form of a hash table that holds the SMR offset for each mapped block. This table is placed in off-chip memory due to limited on-chip memory resources. When translating the address for an external block, the block address is used to locate the corresponding entry in the indirect table, which is used as a prediction for its mapping in the SMR. The mapping table entry indicated is used to determine whether that external block is a hit or miss in the SMR. If we miss, we revert to the first approach where the first several bits of the external block are used to make an accurate decision.

Figure 6.6 illustrates this process. The yellow block needs only one check from the indirect table to verify it is already mapped in SMR and the offset of the mapped SMR is 0x2. The blue block needs two checks, first from the indirect table and second from the first several bits of the blue box in external memory. The offset of the mapped SMR of the blue block is 0x4.

6.2.5 Reducing Address Translation Overheads

In the basic ATHENA design, every off-chip memory access involves a function call into the SL. However, function calls involve significant overhead for saving context. In the case that the block is already in the SMR and ready for access, we should favor a more streamlined approach that offers rapid address translation and minimal work to validate the mapping. In this section, we discuss optional optimizations to reduce the runtime overhead for address translation for the cases where the block is already mapped.

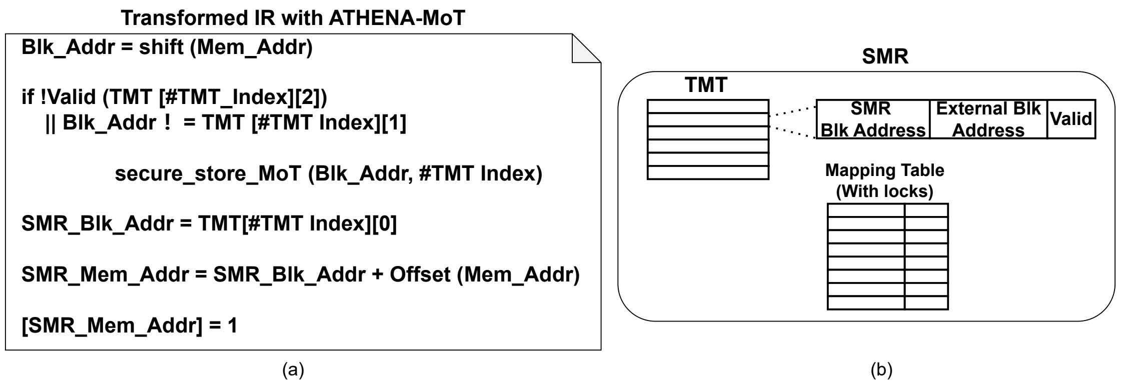
Memoization of Translation

Some memory instructions are very likely to access different addresses in the same external block successively, for example those in a loop that are striding through an array. For such instructions, if we remember the mapped SMR block on its first execution, the following translations can be calculated by adding its offset in the block to the base SMR block address. Based on this observation, we propose **ATHENA-Memoization of Translation (ATHENA-MoT)**.

ATHENA-MoT provides lightweight translation for memory instructions that are likely to repeatedly access the same external block, as often seen in loops. MoT reserves space in the SMR for an instruction's history of translation, called the **Translation Memoization Table (TMT)**. As shown in Figure 6.7 (b), each entry of TMT consists of three parts: SMR block address, its corresponding external block address and a valid specifier.

ATHENA-MoT requires the compiler to take some extra steps to translate a memory access. Figure 6.7 (a) shows an example of a transformed memory access with MoT. It first gets the external block address by shifting the memory address. Then, the block address is compared with the external block address recorded in the TMT for that instruction. The compiler gives each instruction in a loop an index called the TMT index, that's how it knows where to look in the TMT. If the requested block address does not match the block address in the TMT, secure_store_MoT is invoked with two arguments: off-chip memory address and the TMT index. Inside function secure_store_MoT, the entry in the TMT is updated with the most recent translation of this memory access. After the TMT entry is updated and proven valid, the memory instruction uses the translation to access memory in the SMR.

One challenge with this approach is that entries may be present in the TMT for a long time, and its possible that an entry's corresponding block in the SMR could be silently evicted due to intervening requests. We do not want to check all TMT entries on each memory instruction to ensure their validity. To prevent a silent eviction of the recorded SMR block in TMT, any SMR block referenced from by a TMT entry is locked to avoid being evicted. To achieve the lock mechanism, the SMR is augmented with an array of lock counters. Every time a SMR block is recored in TMT, the associated lock counter is incremented by one. When an TMT entry is initialized or overwritten, the lock counter of the old recorded SMR block is decremented by one. A SMR block can be safely evicted if its



Transformed IR with ATHENA-MoT

```
Blk_Addr = shift (Mem_Addr)

if !Valid (TMT [#TMT_Index][2])
  || Blk_Addr !  = TMT [#TMT Index][1]

          secure_store_MoT (Blk_Addr, #TMT Index)

SMR_Blk_Addr = TMT[#TMT Index][0]

SMR_Mem_Addr = SMR_Blk_Addr + Offset (Mem_Addr)

[SMR_Mem_Addr] = 1
```

Figure 6.7: Memoization of Translation (MoT) (a) Example of Transformed Codes with MoT. (b) Structure of Translation Memorization Table (TMT)

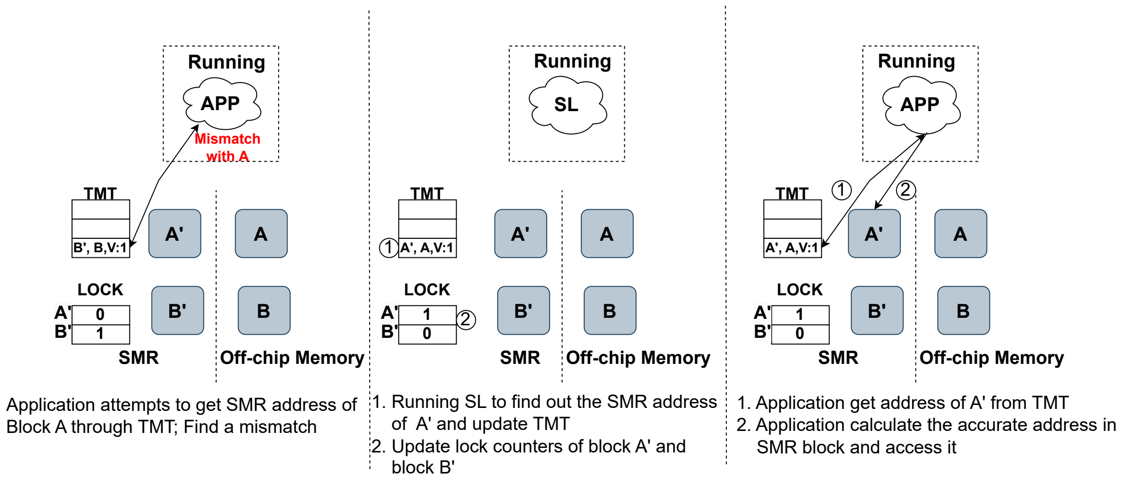

Running
APP
Mismatch with A
TMT
B', B,V:1
A'
A
LOCK
A' 0
B' 1
B'
B
SMR
Off-chip Memory
Application attempts to get SMR address of Block A through TMT; Find a mismatch
Running
SL
TMT
1 A', A,V:1
LOCK
A' 1
B' 0
2
SMR
Off-chip Memory
1. Running SL to find out the SMR address of A' and update TMT
2. Update lock counters of block A' and block B'
Running
APP
1
2
TMT
A', A,V:1
LOCK
A' 1
B' 0
SMR
Off-chip Memory
1. Application get address of A' from TMT
2. Application calculate the accurate address in SMR block and access it

Figure 6.8: Process of Memoization of Translation (MoT)

lock counter is zero. Figure 6.8 illustrates the process of ATHENA-MoT.

Athena-MoT can be applied to any memory instruction but will bring the best performance when there is a high chance of block re-use by successive accesses. We only apply this transformation to memory statements in loops. Considering memory storage consumed by TMT, we implement fixed number of entries in TMT and these entries can be recycled and shared by different memory instructions. Overall, this approach can reduce the number of instructions needed on a translation, on average, from 22 to 8.

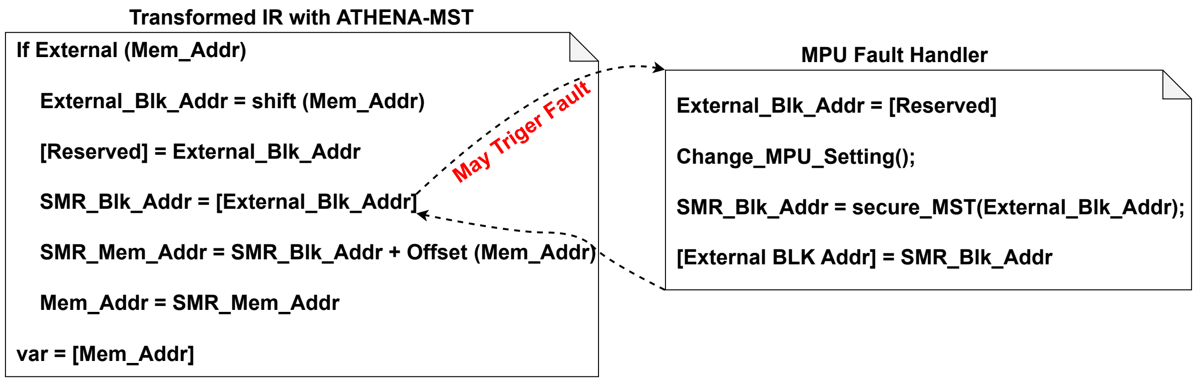

Transformed IR with ATHENA-MST
If External (Mem_Addr)
External_Blk_Addr = shift (Mem_Addr)
[Reserved] = External_Blk_Addr
SMR_Blk_Addr = [External_Blk_Addr]
SMR_Mem_Addr = SMR_Blk_Addr + Offset (Mem_Addr)
Mem_Addr = SMR_Mem_Addr
var = [Mem_Addr]
May Triger Fault
MPU Fault Handler
External_Blk_Addr = [Reserved]
Change_MPU_Setting();
SMR_Blk_Addr = secure_MST(External_Blk_Addr);
[External BLK Addr] = SMR_Blk_Addr

Figure 6.9: Transformed Codes with MPU Silent Translation (MST)

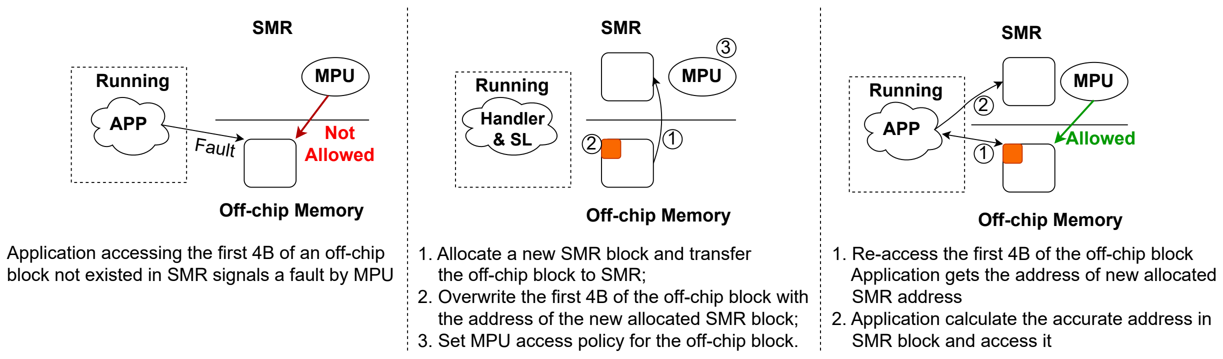

SMR
Running
APP
MPU
Fault
Not Allowed
Off-chip Memory
Application accessing the first 4B of an off-chip block not existed in SMR signals a fault by MPU
SMR
Running
Handler & SL
MPU
3
2
1
Off-chip Memory
1. Allocate a new SMR block and transfer the off-chip block to SMR;
2. Overwrite the first 4B of the off-chip block with the address of the new allocated SMR block;
3. Set MPU access policy for the off-chip block.
SMR
Running
APP
MPU
2
1
Allowed
Off-chip Memory
1. Re-access the first 4B of the off-chip block Application gets the address of new allocated SMR address
2. Application calculate the accurate address in SMR block and access it

Figure 6.10: Process of MPU Silent Translation (MST)

MPU Silent Translation

ATHENA-MoT approach works when the same memory statement accesses the same SMR block successively. For other access patterns, ATHENA-MoT still requires the SL to translate the address with the overhead of a function call. To have a more robust mechanism to enable fast translation for general access patterns, we propose another optimization approach called **ATHENA-MPU Silent Translation (ATHENA-MST)**. The key idea is to streamline translation for the common case when the block is already present in the SMR. The compiler will insert a streamlined set of instructions that we expect to succeed. However, if the translation should fail because the block is not yet in the SMR, the MPU detects it and triggers a fault handler that invokes the slower SL translation process. By having the MPU detect the failure, we keep the generated code shorter.

This approach relies on the hardware Memory Protection Unit (MPU) to detect whether an off-processor block is already mapped to SMR, and it signals a fault when the block might not be mapped yet. This is accomplished by the following. Any block not mapped in the SMR is marked as inaccessible by the MPU. For blocks that are mapped, they are accessible by the MPU, and the block's translated address in the SMR is stored in the first 4 bytes of the block. Hence to perform a translation, the first 4 bytes of the off-chip block are accessed. If it succeeds, it obtains its translation, but if it is blocked by the MPU, MPU fault handler falls back to the SL to decrypt and transfer the corresponding off-processor block to SMR.

To make this work, the MPU fault handler must collaborate with the generated code to perform the translation in the event of failure. Since there is no means for passing arguments or obtaining return values from the MPU handler, we need to construct a way to communicate the faulting address to the MPU fault handler and to allow the handler to return the translated address. We reserve 4B space in the trusted on-chip memory to store the potentially faulting address before the memory statement. The handler will read the fault address from the reserved 4B space whenever it's invoked. The translated address is stored in the first 4B of the off-chip block. After returning from the handler, the same instruction that triggers the fault is re-executed with the same off-chip address, but now the access policy has changed inside the handler. And, it will read the mapped address which can be used to calculate the location for the access.

Figure 6.9 shows one example of transformed memory statement using ATHENA-MST and Figure 6.10 illustrates the process of ATHENA-MST. However, the aforementioned ATHENA-MST approach has two side effects: 1) an SMR memory block will be updated in application without changing the dirty bit. 2) If we transform a memory statement already having internal memory address, the application will fail. To resolve the first side effect, we only apply ATHENA-MST on memory load statement. To resolve the second issue, we add extra statements to detect whether the accessed memory block is an internal or external memory and one if statement to take different actions based on the result.

When we apply ATHENA-MST, we apply it to all memory load statements in a program. Overall, this approach can reduce the number of instructions needed on a translation, on average, from 22 to 13.

6.2.6 Optimized Encryption

Currently, ATHENA deploys counter-mode encryption with AES algorithm. To minimize the AES encryption overhead, we implement a lightweight encryption mechanism adapted from counter-mode encryption, similar to an idea proposed in (54). (54) proposes calculating OTP by calculating two OTPs using AES with address-only input and counter-only input, and achieving the final OTP by multiplying the address-only OTP and counter-only OTP together. ATHENA adopts the idea proposed in (54) and we call it Decoupled-AES (DAES). By using regular memory encryption mechanism with AES, if the size of a SMR block is larger than AES encryption granularity, to encrypt/decrypt it, we need to generate multiple OTPs. For example, to encryption a SMR block sized with 256B using a typical 16B-size AES block, 16 OTPs need to be calculated. By adopting DAES, for one SMR block, we calculate multiple counter-only OTPs (each OTP is calculated by incrementing a base counter value) and single address-only OTP. We observe that some SMR blocks are read-only or have the same update frequency in off-processor memory. To reduce the number of calculations of OTPs to encrypt/decrypt a SMR block, we cache a few sets of counter-only OTPs calculated with different base counter values inside internal memory. If the one SMR block is associated with a base counter value hit in the cache, we can read the set of counter-only OTPs and only need to calculate a single address-only OTP.

6.2.7 Security Analysis

During compilation, ATHENA checks every memory access issued in the application and transform them when needed. Memory footprint of SL operations is always kept inside the internal SRAM, including cryptography keys. We also use MPU as a safe net to detect and intercept off-chip memory reads/writes of application data issued directly from applications. MPU access policy of application data in off-chip memory is set to be allowed in

SL whenever data movement is needed between on-chip memory and off-chip memory. After SL finished data movement, MPU access policy of off-chip memory application data is reset to be non-readable and non-writable. One exception is the MST approach. MST allows part of off-chip application data blocks to be readable by the application. However, this won't breach the security requirement. In MST, application only attempts to read the first 4 bytes of a transfer block and if it is allowed by MPU, it implys that the corresponding off-chip memory block is already mapped in SMR and the first 4 bytes is the block's mapped block address in SMR and will only be used to calculate the accurate SMR address.
Direct Memory Access (DMA): To transfer data using DMA, we need to first configure DMA with source address and destination address through Hardware Abstraction Layer (HAL). Therefore, to avoid ATHENA bypassed by DMA, SL calls can be added manually in HAL to securely drive DMA.

6.3 Evaluation

6.3.1 Experiment Setup

We evaluate ATHENA on STM32F769I-EVAL board(52) with an ARM Cortex-M7. The STM32F-769NI MCU comes with 2MB Flash memory and 512KB internal RAM. It supports eight MPU regions and each region supports 8 subregions. To enable accessibility of a subregion, the region containing the subregion needs to be enabled, and its corresponding subregion bit also needs to be enabled. We test the 5 ML benchmarks listed in Table6.1. The model of each benchmark is stored and encrypted in the off-chip SDRAM. By default, for each benchmark, we test the execution time of a single inference, and before starting the inference, we evict all blocks from the SMR. We use the ARM GCC compiler toolchain to transform the benchmarks. All benchmarks are compiled with Optimization Level 2 (O2). The default SMR size is 128kB. The default SMR block size is 256B. Table 6.2 lists our detailed evaluation setup. We use the baseline design described in Section 6.1 as our point of comparison.

6.3.2 Overall Performance

Figure 6.11 compares the overall speedup of ATHENA, ATHENA-MoT and ATHENA-MST over the baseline both with and without DAES. Without DAES, ATHENA, ATHENA-MoT and ATHENA-MST achieve averaged normalized execution times of 0.23×, 0.19× and 0.13×. With DAES, ATHENA, ATHENA-MoT and ATHENA-MST achieve averaged normalized

Table 6.2: Experiment Setup

Evaluation Board	STM32F769I-EVAL
Flash Memory (On-chip)	2MB
RAM Memory (On-chip)	512kB
SMR Size (On-chip)	128kB (Part of RAM Memory)
Off-chip Memory	8M×32bit SDRAM
SMR Block Size	256B
Compiler	GNU Arm Embedded Toolchain 10.3
Encryption Algorithm	AES_CTR
Counter Size / SMR block	8 Bytes
Number of Reserved Entries for TMT	20
Number of MPU Regions Used by MST	4
Size of MPU Sub-Region Size	Same as SMR Block Size
Cryptography Library	wolfSSL

execution times of 0.16×, 0.13× and 0.09×. All these 5 benchmarks have good off-chip data locality that enables high re-usability of on-chip cached counter-only OTPs, therefore, for all benchmarks, the design using DAES achieves more speedup compared with the same design without DAES. However, for the PD and IC benchmarks, using DAES does not show an obvious speedup mainly because encryption overhead is a small amount of the overall execution time even in the design without DAES. Hence, in much of the rest of our analysis, we compare amongst designs using DAES.

ATHENA-MST achieves more speedup than ATHENA-MoT and ATHENA, mainly because ATHENA-MST significantly reduces the frequency of SL involvement, as shown in Table 6.3. The IC benchmark is an exception in that ATHENA-MST incurs more SL calls than ATHENA-MoT. This is because ATHENA-MST still transforms some memory statements using basic ATHENA when ATHENA-MST is not suitable for them (i.e., memory store statements). Most of the SL calls in IC with ATHENA-MST are caused by using basic ATHENA, and the majority of these SL calls are accessing on-chip memory, which results in an immediate return with an on-chip address. In the case of IC, ATHENA-MST is still able to eliminate unnecessary SL calls for memory statements accessing off-chip memory. Therefore, ATHENA-MST still achieves better performance for IC even though it has more SL calls.

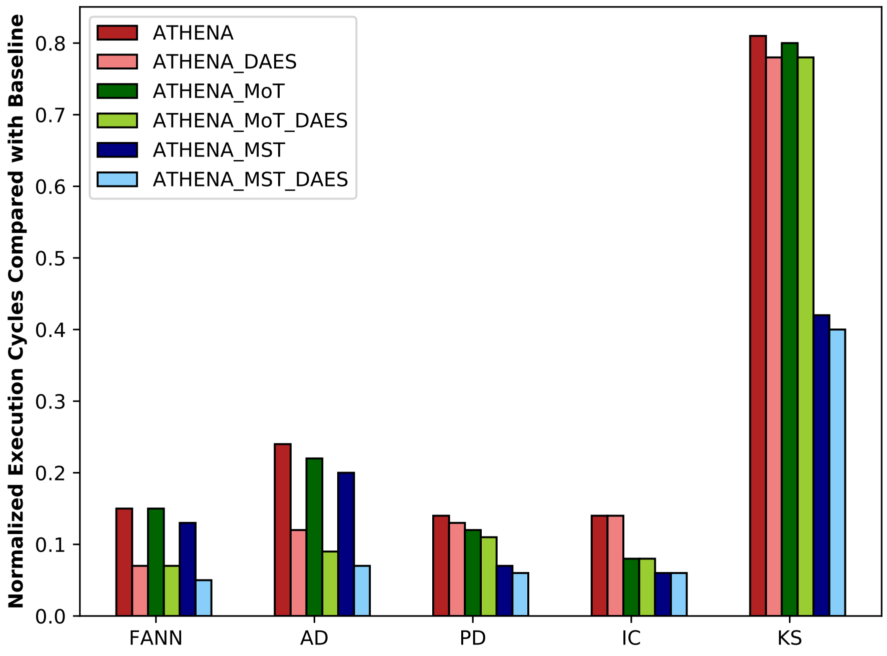


Figure 6.11: Speedup of ATHENA, ATHENA-MoT and ATHENA-MST with/without DAES

Table 6.3: Counting Calls into Secure Library

	Number of Calls into Secure Library		
Benchmark	ATHENA	ATHENA-MoT	ATHENA-MST
FANN	70177	22455	2388
AD	117698	21098	2928
PD	4316279	1216696	690301
IC	5446616	338450	365082
KS	1335220	455961	204620

6.3.3 Sensitivity to secure SMR block size

We study the performance of ATHENA with DAES over different SMR block sizes of 256B, 512B, and 1024B. As shown in Figure 6.12, for the case of ATHENA with DAES, AD achieves more speedup with larger block sizes mainly because larger block sizes prefetch data and save some routine preparation for cryptography operations and data movements. For other benchmarks such savings are negligible, and they still achieve similar speedup using other block sizes.

With larger SMR block sizes, ATHENA-MoT with DAES achieves better speedups, reducing execution time to an average of 0.13×, 0.11× and 0.10× for SMR block size of 256B, 512B and 1024B respectively. This is mainly because a larger SMR block size reduces the frequency of SL involvement. Although ATHENA-MST also reduces the frequency of SL involvement with larger SMR block size, it achieves a similar speedup for other SMR block sizes. ATHENA-MST has significantly reduced the frequency of SL involvement even with a small SMR block size of 256B, and therefore further reducing the frequency of SL involvement with larger SMR block size achieves negligible speedup. We also observed that using larger SMR block size may cause over-fetching of application data, which results in redundant cryptography operation. However, with the implementation of DAES, the overhead of redundant cryptography operation is negligible.

6.3.4 Sensitivity to SMR Size

In this section, we study the performance of ATHENA over different SMR size of 64KB, 128KB and 256KB, both with and without warming-up the SMR. Figure 6.13 shows the performance without warming up. Without warming up, in most benchmarks, the performance gap is negligible for different block sizes of 64KB, 128KB and 256KB. This is mainly because increasing SMR size does not significantly reduce the execution overhead of encryption. The number of transferred data size between SMR and off-chip memory is similar for different SMR sizes. The frequency of calling SL stays the same for other SMR sizes.

Figure 6.14 shows the performance with warming up. In this experiment, we execute 11 inferences and the first inference warms up the SMR. With warm-up, increasing SMR size improves performance in some scenarios because the amount of transferred data between the SMR and off-chip memory is significantly reduced by adopting a larger SMR. In the case of AD and PD, ATHENA-MoT using an SMR size of 256KB reduces the transferred data size. Such lower transferred data size in ATHENA-MoT stems from the implemented lock

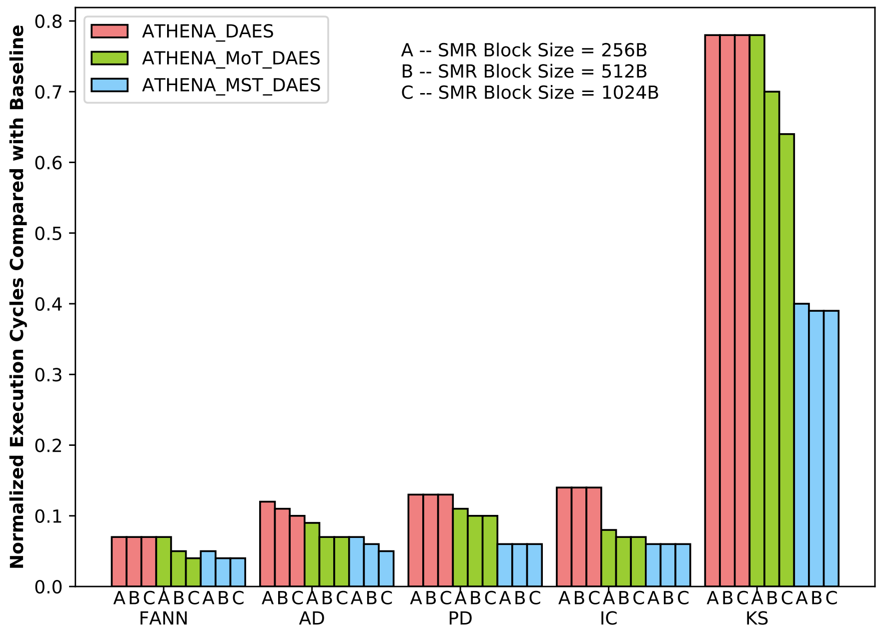


Figure 6.12: Speedup of ATHENA, ATHENA-MoT and ATHENA-MST with DAES for SMR Block Size of 256B, 512B, 1024B

mechanism, which prevents some frequently accessed blocks from being evicted.

6.3.5 Comparison to Non-secure Design

In this section, we compare the performance overhead of ATHENA over non-secure applications. Compared with non-secure applications, the performance overhead of ATHENA mainly comes from three aspects: 1) checking and redirecting memory reference, 2) encryption operations, and 3) overheads of compiler transformation. Without DAES, ATHENA, ATHENA-MoT and ATHENA-MST achieve an averaged slowdown of 10.35×, 8.64× and 6.12×. With DAES, ATHENA, ATHENA-MoT and ATHENA-MST achieve an averaged execution overhead of 7.52×, 6× and 3.97×. As shown in Figure 6.15, comparing all scenarios, ATHENA-MST with DAES achieves the least slowdown.

6.4 Related Works

Some previous work proposes using software to provide or optimize memory encryption. Similar to ATHENA, one previous work (6) on software-based off-chip memory protection

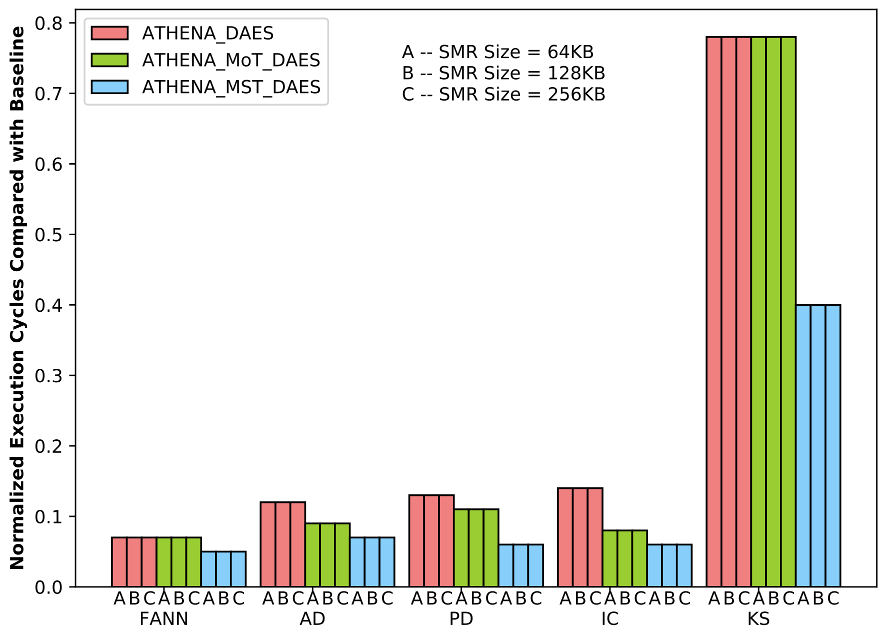


Figure 6.13: Normalized execution cycles of ATHENA, ATHENA-MoT and ATHENA-MST with DAES for SMR Size of 64KB, 128KB and 256KB

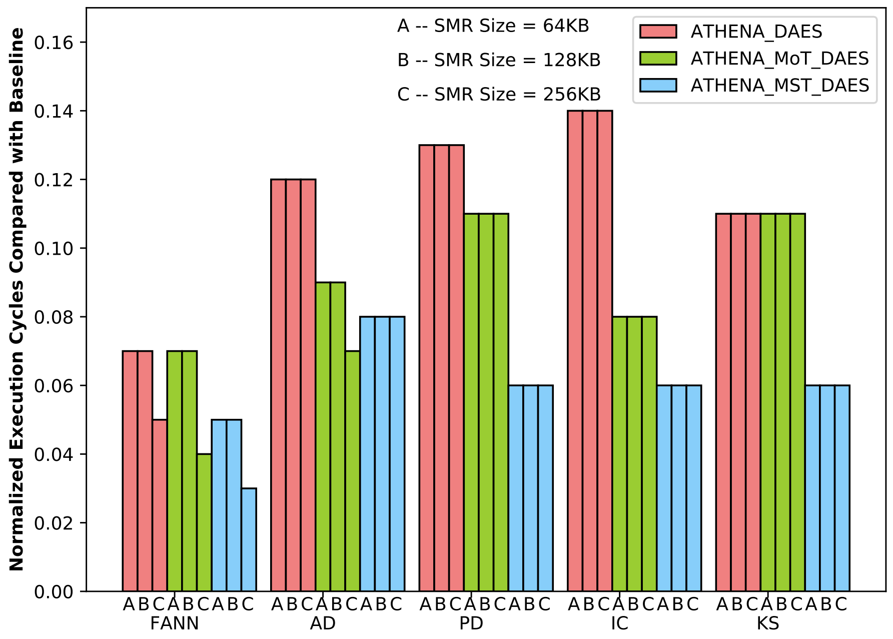


Figure 6.14: Normalized execution cycles of ATHENA, ATHENA-MoT and ATHENA-MST with DAES for SMR Size of 64KB, 128KB and 256KB After Warming-up

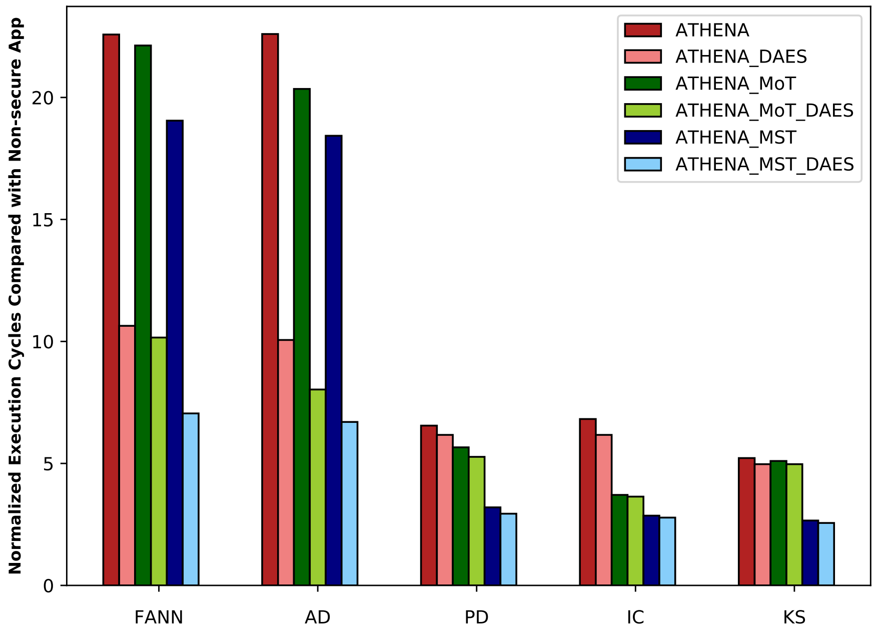


Figure 6.15: Performance Overhead of ATHENA, ATHENA-MoT and ATHENA-MST over Non-Secure Application

also leverages a small on-chip scratchpad memory as trusted on-chip memory region and securely moves data between off-chip memory and on-chip scratchpad memory through software-based encryption/decryption. However, this work requires an MMU paging mechanism to direct data movement between on-chip and off-chip memory, which makes it unsuitable for some embedded systems without an MMU. The other work (43) proposes a compiler-assisted approach to improve memory encryption for embedded systems by reducing off-chip accesses of counter blocks. This work still needs dedicated memory encryption hardware to detect data movement between on-chip/off-chip memory and execute memory encryption operation. In contrast, ATHENA only needs a small on-chip memory to store the decrypted data, leverages an MPU as a "safety net", and proposes optimizations for performance.

Some previous works (35; 18; 11; 46) consider how to improve on-chip memory usage in embedded systems. These works try to maximize re-use of data in on-chip memory through static allocation, dynamic allocation and data movement. ATHENA does not prevent the adoption of these works and such works can be combined with ATHENA to further improve its performance.

6.5 Summary

In this chapter, we propose ATHENA, a novel software-based mechanism to provide application-transparent data confidentiality for embedded systems with minimal hardware requirements (a small on-chip memory region and MPU). ATHENA consists of a static transformation tool implemented with GCC Compiler to automatically modify applications and a run-time secure library (SL) to dynamically and efficiently manage on-chip memory resources and securely transfer and map data between on-chip and off-chip memory. Based on our evaluation, ATHENA reduces the execution time to an average of $0.13\times$, compared with an intuitive design that does not have mechanism to enable reusing decrypted plaintext.

CHAPTER

7 CONCLUSION

In this thesis, we presented hardware and software mechanisms to provide efficient secure memory for Persistent Memory and embedded systems under realistic hardware constraints. Three hardware solutions are proposed to solve challenges of secure Persistent Memory under two kinds of resource limitation: 1) Limited on-chip battery power/size; 2) Absence of extra bits in the memory interface. Dolos and Horus allow fast and secure data persistence without significantly increasing on-chip power. Thoth enables fast persistence of secure metadata without leveraging extra bits in memory interface. In this thesis, we also propose one software solution, Athena, with the support of a compiler to provide memory security for embedded systems without dedicated hardware for memory encryption.

REFERENCES

[1] "Enabling Intel Optane DC Persistent Memory on Lenovo ThinkSystem Servers," "https://lenovopress.com/lp1167.pdf", [Online; accessed 07-November-2021].

[2] "Intel hardware sheild – intel total memory encryption," [Online; accessed 09-November-2021]. [Online]. Available: "https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf"

[3] M. Alshboul, P. Ramrakhyani, W. Wang, J. Tuck, and Y. Solihin, "Bbb: Simplifying persistent programming using battery-backed buffers," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 111–124.

[4] M. Alwadi, K. Zubair, D. Mohaisen, and A. Awad, "Phoenix: Towards ultra-low overhead, recoverable, and persistently secure nvm," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020.

[5] AMD. (2021) Amd memory encryption. [Online; accessed 7-Nov-2021]. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[6] G. Andrade, D. Lee, D. Kohlbrenner, K. Asanović, and D. X. Song, "Software-based off-chip memory protection for risc-v trusted execution environments," 2020.

[7] ARM. Memory protection unit. [Online; accessed 28-March-2024]. [Online]. Available: https://developer.arm.com/documentation/ddi0439/b/Memory-Protection-Unit

[8] Arm. (2021) Arm confidential compute architecture. [Online; accessed 7-Nov-2021]. [Online]. Available: https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture

[9] ARM. (2021) Learn the architecture - realm management extension. [Online; accessed 28-March-2024]. [Online]. Available: https://developer.arm.com/documentation/den0126/latest/

[10] ——. (2024) Tightly-coupled memory. [Online; accessed 28-March-2024]. [Online]. Available: https://developer.arm.com/documentation/ddi0338/g/level-one-memory-system/tightly-coupled-memory

[11] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 1, no. 1, p. 6–26, nov 2002. [Online]. Available: https://doi.org/10.1145/581888.581891

[12] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 104–115.

[13] 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*, ser. ASPLOS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 263–276. [Online]. Available: https://doi.org/10.1145/2872362.2872377

[14] A. Awad, S. Suboh, M. Ye, K. A. Zubair, and M. Al-Wadi, “Persistently-secure processors: Challenges and opportunities for securing non-volatile memories,” in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019, pp. 610–614.

[15] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair, “Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 104–115.

[16] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, U. Thakker, A. Torrini, P. Warden, J. Cordaro, G. D. Guglielmo, J. Duarte, S. Gibellini, V. Parekh, H. Tran, N. Tran, N. Wenxu, and X. Xuesong, “Mlperf tiny benchmark,” 2021. [Online]. Available: https://arxiv.org/abs/2106.07597

[17] 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,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[18] D.-W. Chang, I.-C. Lin, Y.-S. Chien, C.-L. Lin, A. W.-Y. Su, and C.-P. Young, “Casa: Contention-aware scratchpad memory allocation for online hybrid on-chip memory management,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 12, pp. 1806–1817, 2014.

[19] Z. Chen, Y. Zhang, and N. Xiao, “Cachetree: Reducing integrity verification overhead of secure nonvolatile memories,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 7, pp. 1340–1353, 2021.

[20] S. Chhabra and D. Durham, “METHOD AND APPARATUS FOR SHARING SECURITY METADATA MEMORY SPACE,” in *United States Patent Application 20200183861*.

[21] S. Chhabra and Y. Solihin, “I-nvmm: A secure non-volatile main memory system with incremental encryption,” *SIGARCH Comput. Archit. News*, vol. 39, no. 3, p. 177–188, Jun. 2011. [Online]. Available: https://doi.org/10.1145/2024723.2000086

[22] T. W. David Kaplan, Jeremy Powell, “AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More,” AMD, Tech. Rep., 10 2021.

[23] J. Frazelle, “Power to the people: Reducing datacenter carbon footprints,” *Queue*, vol. 18, no. 2, p. 5–18, apr 2020. [Online]. Available: https://doi.org/10.1145/3400899.3402527

[24] A. Freij, S. Yuan, H. Zhou, and Y. Solihin, “Persist level parallelism: Streamlining integrity tree updates for secure persistent memory,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 14–27.

[25] A. Freij, H. Zhou, and Y. Solihin, “Bonsai merkle forests: Efficiently achieving crash consistency in secure persistent memory,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1227–1240. [Online]. Available: https://doi.org/10.1145/3466752.3480067

[26] S. Gueron, “A memory encryption engine suitable for general purpose processors,” Cryptology ePrint Archive, Report 2016/204, 2016, https://ia.cr/2016/204.

[27] X. Han, J. Tuck, and A. Awad, “Dolos: Improving the performance of persistent applications in adr-supported secure memory,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1241–1253. [Online]. Available: https://doi.org/10.1145/3466752.3480118

[28] M. Hoseinzadeh, M. Arjomand, and H. Sarbazi-Azad, “Reducing access latency of mlc pcms through line striping,” in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 277–288.

[29] Intel. (2020) Build persistent memory applications with reliability availability and serviceability. [Online; accessed 7-March-2021].

[30] ——, “eADR: New Opportunities for Persistent Memory Applications,” https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021, [Online; accessed 7-March-2022].

[31] ——, “Intel Software Guard Extensions (Intel SGX),” https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html, 2021, [Online; accessed 7-March-2022].

[32] ——, “Does Intel® Server System M50CYP Support eADR (Enhanced Asynchronous DRAM Refresh)?” https://www.intel.com/content/www/us/en/support/articles/000088236/server-products/single-node-servers.html, 2022, [Online; accessed 7-March-2022].

[33] ——. (2022) Persistent memory development kit. https://pmem.io/pmdk/. [Online; accessed 7-March-2022].

[34] F. Ishibashi, “Introducing Optanedc persistent memory,” [Online; accessed 07-November-2021]. [Online]. Available: "http://www.ipsj.or.jp/sig/os/index.php?plugin=attach&refer=ComSys2019&openfile=ComSys2019-IntelDCPMMver1.0.pdf"

[35] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, “A compiler-based approach for dynamically managing scratch-pad memories in embedded systems,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 2, pp. 243–260, 2004.

[36] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, “Keystone: An open framework for architecting trusted execution environments,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387532

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[38] G. Lim, D. Kang, and Y. I. Eom, “Thread evolution kit for optimizing thread operations on ce/iot devices,” *IEEE Transactions on Consumer Electronics*, vol. 66, no. 4, pp. 289–298, 2020.

[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)*, 2018, pp. 310–323.

[40] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, “Janus: Optimizing memory and storage support for non-volatile memory systems,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 143–156.

[41] Micron. (2021) Introducing micron® ddr5 sdram: More than a generational update. [Online; accessed 7-Nov-2021]. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/white-paper/ddr5_more_than_a_generational_update_wp.pdf?la=en

[42] S. Mittal, J. S. Vetter, and D. Li, “Lastingnvcache: A technique for improving the lifetime of non-volatile caches,” in *2014 IEEE Computer Society Annual Symposium on VLSI*, 2014, pp. 534–540.

[43] V. Nagarajan, R. Gupta, and A. Krishnaswamy, “Compiler-assisted memory encryption for embedded processors,” ser. HiPEAC’07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 7–22.

[44] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An analysis of persistent memory use with whisper,” *SIGARCH Comput. Archit. News*, vol. 45, no. 1, p. 135–148, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3093337.3037730

[45] S. Nissen. (2023) Fast artificial neural network library. [Online; accessed 28-March-2023]. [Online]. Available: http://leenissen.dk/fann/wp/

[46] P. Panda, N. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proceedings European Design and Test Conference. ED TC 97*, 1997, pp. 7–11.

[47] 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 *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 183–196.

[48] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, "Morphable counters: Enabling compact integrity trees for low-overhead secure memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 416–427.

[49] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 454–465.

[50] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient stt-ram caches," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 50–61.

[51] M. Soltani, M. Ebrahimi, and Z. Navabi, "Prolonging lifetime of non-volatile last level caches with cluster mapping," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, 2016, pp. 329–334.

[52] ST. (2024) Evaluation board with stm32f769ni mcu. [Online; accessed 22-Feb-2024]. [Online]. Available: https://www.st.com/en/evaluation-tools/stm32f769i-eval.html

[53] S. Van Doren, "Abstract - hoti 2019: Compute express link," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2019, pp. 18–18.

[54] X. Wang, D. Talapkaliyev, M. Hicks, and X. Jian, "Self-reinforcing memoization for cryptography calculations in secure memory systems," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 678–692.

[55] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 496–508.

[56] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 179–190.

[57] B. Yang, S. Mishra, and R. Karri, "A high speed architecture for galois/counter mode of operation (gcm)," *IACR Cryptol. ePrint Arch.*, vol. 2005, p. 146, 2005.

[58] F. Yang, Y. Chen, H. Mao, Y. Lu, and J. Shu, "Shieldnvm: An efficient and fast recoverable system for secure non-volatile memory," *ACM Trans. Storage*, vol. 16, no. 2, May 2020. [Online]. Available: https://doi.org/10.1145/3381835

[59] F. Yang, Y. Lu, Y. Chen, H. Mao, and J. Shu, "No compromises: Secure nvm with crash consistency, write-efficiency and high-performance," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[60] 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)*, 2018, pp. 403–415.

[61] S. Yoo, D. Kim, Y. M. Koo, S. K. Wooju Jeong, H. Shim, W.-J. Lee, B. S. Lee, S. Lee, H. Choi, H. D. Lee, T. Kim, and M.-H. Na, "Structural and device considerations for vertical cross point memory with single-stack memory toward cxl memory beyond 1x nm 3dxp," in *2022 IEEE International Memory Workshop (IMW)*, 2022, pp. 1–4.

[62] V. Young, P. Nair, and M. Qureshi, "Deuce: Write-efficient encryption for non-volatile memories," *ACM SIGPLAN Notices*, vol. 50, pp. 33–44, 05 2015.

[63] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 421–432.

[64] J. Zhou, A. Awad, and J. Wang, "Lelantus: Fine-granularity copy-on-write operations for secure non-volatile memories," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 597–609.

[65] K. A. Zubair and A. Awad, "Anubis: Ultra-low overhead and recovery time for secure non-volatile memories," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 157–168.

[66] K. A. Zubair, S. Gurumurthi, V. Sridharan, and A. Awad, "Soteria: Towards resilient integrity-protected and encrypted non-volatile memories," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1214–1226. [Online]. Available: https://doi.org/10.1145/3466752.3480066

[67] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 442–454.

[68] P. Zuo, Y. Hua, and Y. Xie, "Supermem: Enabling application-transparent secure persistent memory with low overheads," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY,

USA: Association for Computing Machinery, 2019, p. 479–492. [Online]. Available: https://doi.org/10.1145/3352460.3358290