Dolos: Improving the Performance of Persistent Applications in ADR-Supported Secure Memory

Xijing Han
North Carolina State University
Raleigh, NC, USA
xhan24@ncsu.edu

James Tuck
North Carolina State University
Raleigh, NC, USA
jtuck@ncsu.edu

Amro Awad
North Carolina State University
Raleigh, NC, USA
ajawad@ncsu.edu

ABSTRACT

The performance of persistent applications is severely hurt by current secure processor architectures. Persistent applications use long-latency flush instructions and memory fences to make sure that writes to persistent data reach the persistency domain in a way that is crash consistent. Recently introduced features like Intel's Asynchronous DRAM Refresh (ADR) make the on-chip Write Pending Queue (WPQ) part of the persistency 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. Our goal in this work is to make it feasible for secure persistent applications to benefit more from the on-chip persistency domain.

We propose Dolos, an architecture that prioritizes persisting data without sacrificing security in order to gain a significant performance boost for persistent applications. Dolos achieves this goal 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. To evaluate our design, we implemented our architecture in the GEM5 simulator, and analyzed the performance of 6 benchmarks from the WHISPER suite. Dolos improves their performance by 1.66x on average.

CCS CONCEPTS

• **Hardware → Memory and dense storage**; • **Security and privacy → Security in hardware**.

KEYWORDS

Memory Security, Merkle Tree, MAC, Persistent memory, Encryption

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480118

ACM Reference Format:
Xijing Han, James Tuck, and Amro Awad. 2021. Dolos: Improving the Performance of Persistent Applications in ADR-Supported Secure Memory. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480118

1 INTRODUCTION

Emerging non-volatile memories (NVMs) are expected to play a major role in future computing systems. Unlike DRAM, NVMs can retain data after power loss events, which enables crash-recoverable workloads that leverage this persistent storage capability. Moreover, NVMs do not require frequent refresh operations as in DRAM, which allows significant idle-power savings. Finally, NVMs feature high capacities and potential for further scalability. For instance, existing NVM products, e.g., Intel Optane DC Persistent Memory (DCPMM)[12], have capacities of 128GB-256GB per memory module. With all these features, NVMs are expected to replace DRAM or augment it as a part of the main memory.

Due to various challenges for securing a trusted supply chain of integrated circuits chips, the move towards computing paradigms where computing systems are not physically under the control of users, and various advancements in hardware trojans and attacks, minimizing the trusted computing base (TCB) to the minimum possible is of utmost importance. In state-of-the-art secure processors, the TCB is constrained to the processor chip boundaries, and thus the integrity and confidentiality of data must be protected when leaving the processor chip boundaries. Even though secure processors are necessary and already adopted for DRAM-based memory, they become even more needed with NVMs due to their data remanence vulnerability. However, integrating secure processors with NVMs is challenging; the implementation of secure processors should account for the recoverability, write endurance, and performance aspects of NVMs[3, 5, 6, 14, 19, 20, 23, 25, 26].

Persistent workloads that leverage the data retention capability of NVMs are expected to explicitly flush its persistent data updates from the processor's volatile caches to the *persistent domain*. In such a persistency model, currently supported in Intel processor, persistent applications frequently execute cacheline flush operations followed by a memory fence to force ordering of persistence operations with following instructions. However, due to the high write latency of NVMs, persisting updates by flushing them all the way to the NVM module can significantly degrade performance[16]. Therefore, recent support from processor vendors extends the persistence domain to include a small write buffer, inside the processor chip, called *write pending queue (WPQ)*[13]. 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 NVM write latency. The support for additional reserved power to flush the internal WPQ buffer is called Asynchronous DRAM Refresh (ADR). ADR support is considered a mandatory platform requirement for systems adopting persistent memories[11]. While ADR could be further extended (e.g., as in enhanced ADR (eADR)) to further reduce the persistence overheads, it comes at the cost of non-standard extensions, high costs, and environment-unfriendly batteries. Thus, we expect most future persistent memory systems to have the standard ADR support which extends the persistent domain to include the processor's WPQ buffer. Due to security and crash consistency reasons, state-of-the-art secure NVM implementations apply all secure metadata updates and operations before inserting the data in the persistence domain, i.e., the data written to WPQ is treated similarly to off-chip data.

The Challenge: Before considering a cacheline persisted, secure NVM controllers need to ensure that all the associated security metadata are updated and crash consistent. Since the WPQ is considered as part of the persistent domain, such metadata updates along with expensive cryptographic operations directly add to the critical path latency of data persistence operations. The overheads of security metadata updates and cryptographic operations by far exceed the NVM write latency, which renders the ADR support ineffective; data persistence operations will incur significant latency before the data reaches the persistence domain. Based on our experiments, persistent workloads encounter an average performance overhead of 52% (up to 61%) compared to an ideal secure NVM system where data is considered persisted immediately after it is flushed from the processor's internal caches, i.e., as in non-secure processors. Prior works[6, 14, 15, 23, 26] discussed the challenges of implementing secure and crash-consistent NVMs. However, all prior works assume that the memory backend security operations should occur before the data to be persisted reaches the persistence domain (WPQ), and hence incur significant performance overheads when used with persistent applications.

Our **key observation** is 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 paper, 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.,[2, 6, 15, 26]), where Dolos minimizes the the latency of the memory front-end operations (insertion into the WPQ), and hence significantly improves persistent workloads' performance.

To evaluate Dolos, we use Gem5 simulator[7], an open-source cycle-level simulator, to run representative persistent workloads from Whisper[16], in addition to in-house developed workloads. On average, Dolos improves the overall performance by 1.66x, compared to the state-of-the-art secure NVM implementations. Moreover, we show how Dolos provides the same security and recoverability guarantees of the state-of-the-art secure NVM implementations.

The rest of the paper is organized as follows. First, in Section 2, we discuss the background topics related to this work. Section 3 describes the motivation of this paper. Section 4 introduces the Dolos design. Evaluation methodology and results are shown in Section 5. Related works are in Section 6. Conclusions are in Section 7.

2 BACKGROUND

In this section, we present the main concepts related to our work.

2.1 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[9], the data is encrypted/decrypted when written/read inside the memory module. Thus, any physical

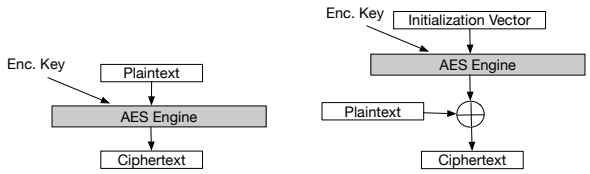
attacks that could successfully acquire the memory module would fail to breach the confidentiality of data. However, malicious implants or hardware trojans in the memory bus or memory slots can observe the memory data not encrypted and hence breach confidentiality. Due to the challenges of ensuring a 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 paper, 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 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. 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 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 paper, we will use counter-mode encryption due to its security and performance advantages of direct 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



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

Figure 1: Difference between direction encryption and counter-mode encryption.

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[8]. 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.

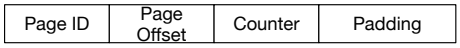


Figure 2: The fields of the initialization vector used in counter-mode encryption.

2.2 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 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 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 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[26]. While the solutions proposed in this work apply to both ToC and MT, we limit the scope of discussion for the rest of the paper 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.

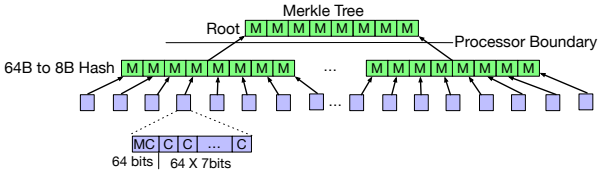


Figure 3: Merkle Tree.

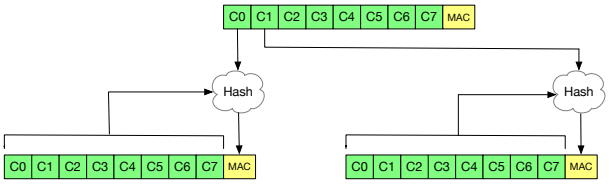


Figure 4: Tree of Counters.

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 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[6, 26]. Therefore, eager MT update is generally used with secure NVMs[6, 10, 26].

Since both encryption counters and data need to have their integrity protected, state-of-the-art solutions leverage a Bonsai Merkle Tree (BMT)[18], 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.

2.3 Persistent Security

To ensure crash recovery of secure NVMs, special handling of security metadata (encryption counters and MT nodes) is required. Thus, there was a large body of work that addressed the recoverability of secure NVMs, commonly referred to as persistent security[4, 6, 10, 14, 23, 26, 27]. Osiris[23] allowed the recovery of encryption counters updated on the volatile counter cache by leveraging Error-Correction Codes (ECC) written with the data. Thus, recovering the counter used for encrypting the data can be done by using ECC as a sanity check for the decryption process. Liu et al.[14] propose an approach that relaxes the atomicity of counter updates for non-persistent data structures. Other works, such as Triad-NVM[6] and Anubis[26] augmented prior crash recovery solutions with the ability to recover integrity trees and reduce the recovery time, respectively. Freij et al.[10] explore mechanisms to speed up MT eager updates through pipelining and break the serialization between various memory writes in terms of MT updates.

In general, all the prior works aim to ensure recoverability, reduce write overheads, and allow fast recovery after crashes. 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 conventional non-secure memories, the same problem exists, due to the long write latency of NVMs, that the application needs to wait for before considering the data persisted. Thus, recent processors provide battery-backed fast write buffers inside the processor chip that can be flushed to the NVM upon a power failure, and thus data persistence operation can finish much faster as it longer needs to wait until the data reaches the NVM. These buffers are called write-pending queues (WPQs), and the standard system feature that enables flushing the WPQ to NVM upon power failure is called Asynchronous DRAM Refresh (ADR)[11]. Unfortunately, current works completely overlook the main purpose of WPQ, which is reducing the latency of the critical path for data persistence operations. Thus, our paper is the first to investigate how the WPQ can be leveraged to reduce the critical path latency of persistence operations in secure NVMs.

Persistent applications utilize the persistency feature of NVM by storing objects in NVM and maintaining a way to reconnect

to stored NVM objects across application runs. When writing persistent applications, one major issue that needs to be considered is data persistency and atomicity of updates[1, 21]. Data persistency mandates flushing the data to be persisted all the way to the persistence domain (e.g., WPQ). While atomicity of updates aims to ensure that either all updates or none finish, which is particularly important when updating a large data structure or related information within a transaction. Current persistence libraries, e.g., Intel's Persistent Memory Development Kit (PMDK)[17], provides users with mechanisms to explicitly persist updates and ensuring they reached the persistence domain through cache flushing and memory fences, respectively.

Unlike conventional applications, where writes can be buffered and not in the critical path of the application, persistent workloads' performance heavily depends on the latency of persistence operations, and they can lead to 24.3% performance overheads on average[16]. Thus, it is important to minimize the overheads of data persistence operations in such workloads.

3 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 5 shows several possible secure processor architectures with NVM. The most basic is shown in Figure 5(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.

The introduction of ADR, bringing the WPQ into an on-chip persistency domain, leads to the the two architectures in Figure 5-b and Figure 5-c. In these architectures, the persistency domain is the whole off-chip memory and the ADR-enabled WPQ. In Figure 5-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 5-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 6 shows a performance comparison when performing the security operation before the WPQ (Fig 5-b) to a hypothetical scheme that allows delaying a security operation until after eviction from the WPQ

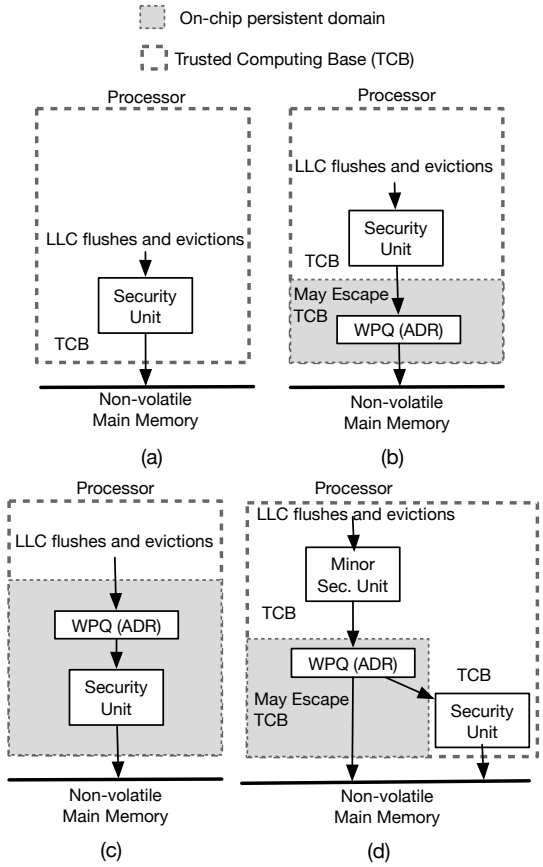


Figure 5: Models of Secure Memory and Secure Persistent Memory: a) Secure Memory Model without ADR, b) Secure Persistent Memory Model that cannot take full advantage of ADR, c) an infeasible approach that requires the persistency domain to subsume the Security Unit, and d) our approach: lower latency Minor Security Unit protects WPQ entries, major Security Unit protects memory.

(Fig 5-c). We use the same setup as described in Section 5 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.

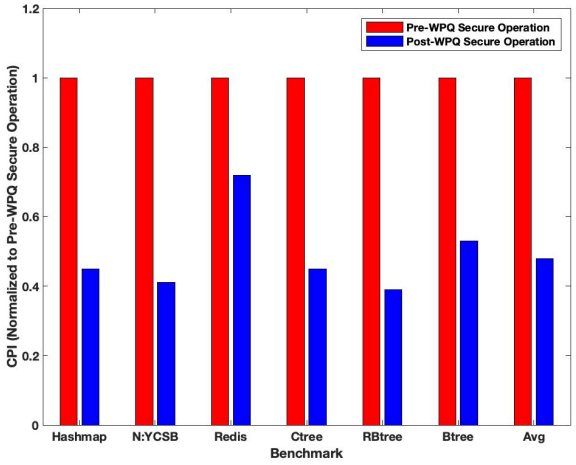


Figure 6: CPI between placing security process before and after WPQ.

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 5-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.

4 DOLOS 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.

4.1 Threat Model

Our threat model is similar to the state-of-the-art work on secure NVM[3, 6, 23, 24, 26]. 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, our solution needs to detect such attacks.

4.2 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 7. 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[6] or Anubis[26]. In this work, Ma-SU uses Anubis as the Ma-SU mechanism for crash consistency of run-time updates.

4.3 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 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

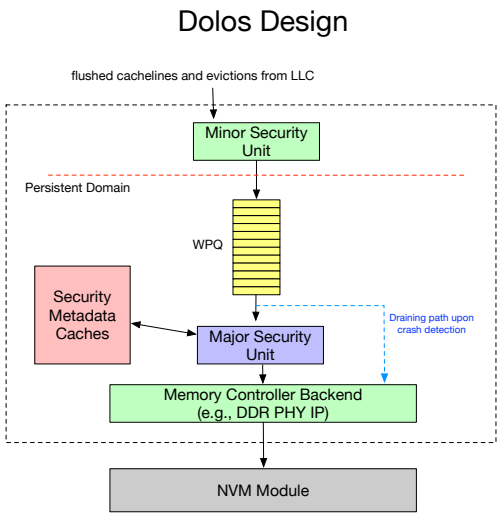


Figure 7: System Overview

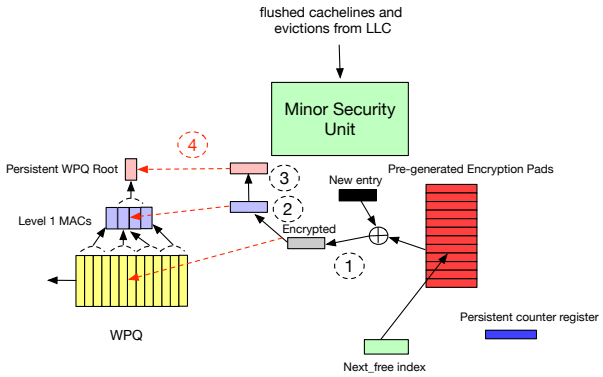


Figure 8: The Full-WPQ-MiSU scheme for Mi-SU.

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 (not confidential as discussed earlier in Section 2) 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***.

As shown in Figure 8, 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).

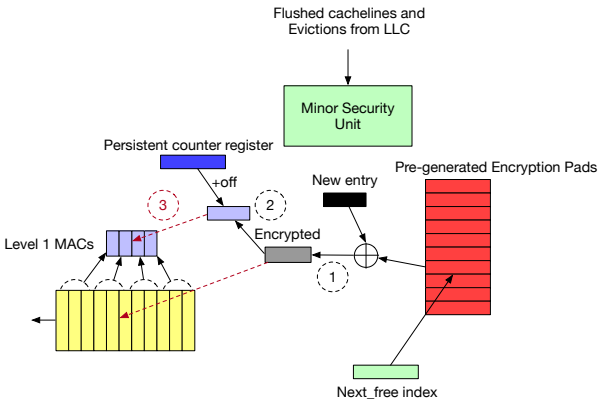


Figure 9: The Partial-WPQ-MiSU scheme for Mi-SU.

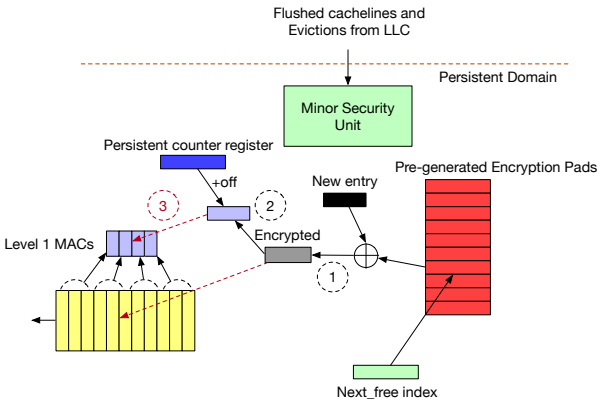


Figure 10: The POST-WPQ-MiSU scheme for Mi-SU.

As shown in Figure 9, 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 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 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 10. 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.

4.4 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[23] and Anubis[26], which will be described in Section 6. 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[26]. 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.

The steps of Ma-SU upon a single WPQ entry are shown in Figure 11. ① Use the next_fetch_index to fetch one WPQ entry,

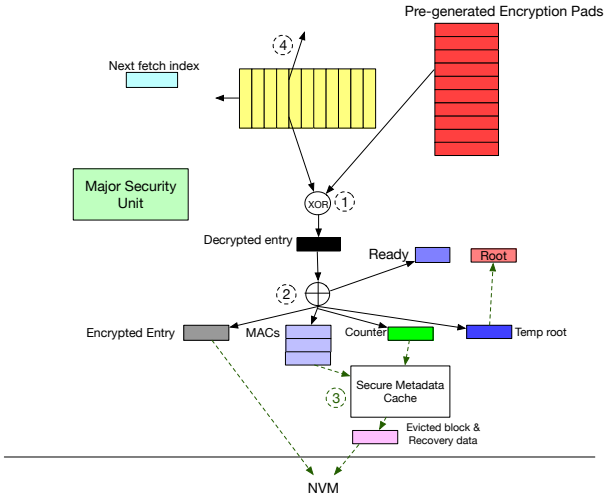


Figure 11: Ma-SU Scheme.

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[26]) 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[26] and Triad-NVM[6]) we assume the counters are recoverable using Osiris[23]. Hence, we adopt AGIT[26] for MT where the root is eagerly updated. For ToC, as shown by prior works[2, 26], eagerly updating the root persistently is insufficient to recover the tree due to inter-level dependencies in SGX-style trees[26]. Therefore, state-of-the-art schemes[2, 26] 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[2] 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.

4.5 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.

4.6 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.

5 EVALUATION

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

Table 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	160X10 Cycles for Eager Update; 160X4 Cycles for Lazy Update
Integrity Tree	8-ary Merkle Tree; 8-ary TOC
Tree Update Policy	Eager Update(Merkle Tree); Lazy Update(TOC)

5.1 Simulation Setup

We use GEM5[7], a cycle-level simulator to evaluate the performance overheads of Dolos. As illustrated in Table 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[16]. 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[26]) and ToC with lazy update (as in Phoenix[2]). For the update of ToC, AES-GCM and conservatively assume parallel AES-GCM engines with a design and latency based on prior work[22], 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[6, 15, 26]. 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[6, 15, 26]. 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[26] – 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[26] 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.

5.2 Dolos Performance

5.2.1 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.

[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 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

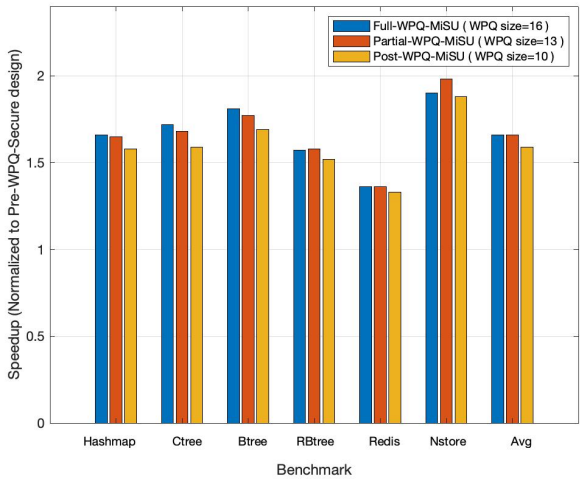


Figure 12: Speedup of Dolos with Full-WPQ-MiSU, Partial-WPQ-MiSU, Post-WPQ-MiSU using Eager Update Scheme(transaction size = 1024B)

Figure 12 shows an average speedup of 1.66x, 1.66x, 1.59x 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 12, 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 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 1.98x and Full-WPQ-MiSU has speedup of 1.90x. 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.

5.2.2 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

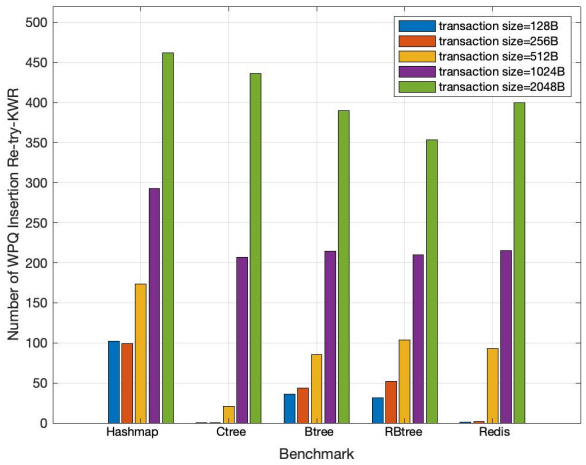


Figure 13: Number of WPQ insertion Re-try-KWR of Dolos with Partial-WPQ-MiSU on single transaction size of 128B, 256B, 512B, 1024B, 2048B

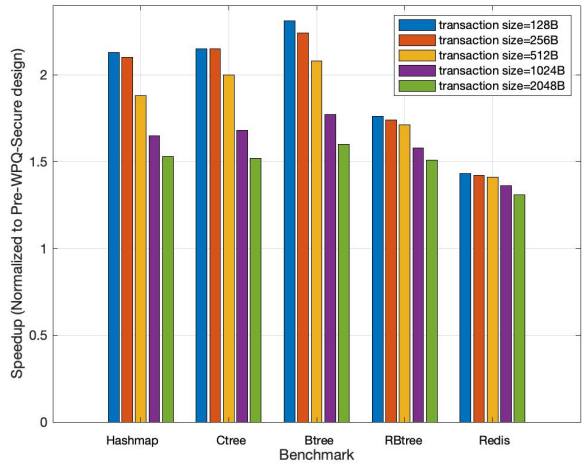


Figure 14: Speedup of Dolos with Partial-WPQ-MiSU on single transaction size of 128B, 256B, 512B, 1024B, 2048B

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

5.3 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 15, the performance of Dolos with Partial-WPQ-MiSU improves when WPQ size increases. Dolos achieves an average speedup of 1.66x, 1.85x, 1.87x and 1.88x 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.

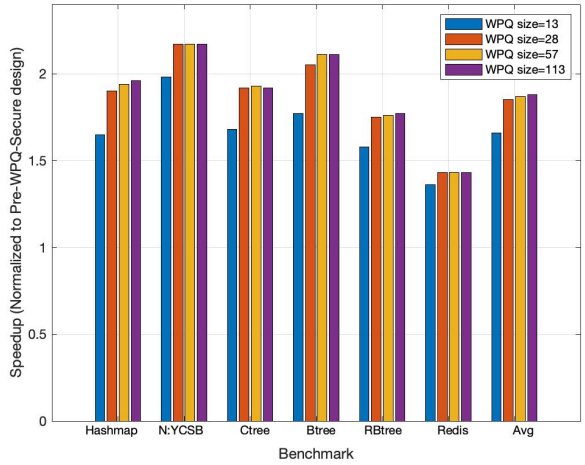


Figure 15: Speedup of Dolos with Partial-WPQ-MiSU on WPQ size of 13, 28, 57, 113 (transaction size = 1024B)

5.4 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[2]. Figure 16 shows an average speedup of 1.044x, 1.079x, 1.071x 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.

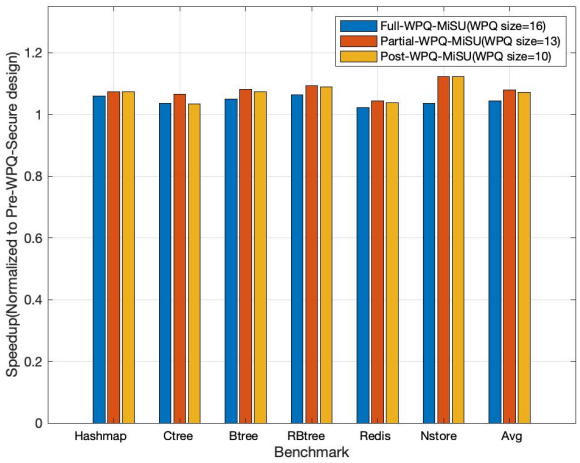


Figure 16: Speedup of Dolos with Full-WPQ-MiSU, Partial-WPQ-MiSU, Post-WPQ-MiSU using Lazy Update Scheme(transaction size = 1024B)

Table 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

5.5 Estimated Overheads of Mi-SU

We estimate the storage overhead and recovery overhead of Mi-SU using a 16-entry WPQ size. Table 3 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).

6 RELATED WORK

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 as discussed in Section 2.3. Prior works[4, 6, 10, 14, 23, 26, 27] propose mechanisms to speed up recovery of security metadata. Osiris[23] 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[23] for more details). Osiris has a long recovery time as it needs to rebuild the whole integrity tree based on recovered counters. Thus, Anubis[26] 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 [15, 19, 24, 27] reduce the overhead of secure NVM in general. Morphable Counters[19] 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[24] propose a scheme to only re-encrypt changed words in a single cacheline, thus reducing flipped bits per writeback. Zuo et al.[27] 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[15] 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[15]). 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.*

7 CONCLUSION

In this paper, 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 1.66x over the state-of-the-art schemes 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.

We hope that Dolos will further facilitate the adoption of secure NVMs and enable efficient execution of persistent workloads in such systems. We also hope that this work opens up a new research direction for efficiently exploiting WPQ in secure NVM implementations.

ACKNOWLEDGEMENT

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA), the Office of Naval Research (ONR), and National Science Foundation (CNS-1717486). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

REFERENCES

[1] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy Persistency: a High-Performing and Write-Efficient Software Persistency Technique. In *2018 ACM/IEEE 45TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE (ISCA)*. 439–51. https://doi.org/10.1109/ISCA.2018.00044

[2] Mazen Alwadi, Kazi Zubair, David Mohaisen, and Amro Awad. 2020. Phoenix: Towards Ultra-Low Overhead, Recoverable, and Persistently Secure NVM. *IEEE Transactions on Dependable and Secure Computing* (2020), 1–1. https://doi.org/10.1109/TDSC.2020.3020085

[3] Amro Awad, Pratyusa Manadhata, Stuart Haber, Yan Solihin, and William Horne. 2016. Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 263–276. https://doi.org/10.1145/2872362.2872377

[4] Amro Awad, Suboh Suboh, Mao Ye, Kazi Abu Zubair, and Mazen Al-Wadi. 2019. Persistently-secure processors: Challenges and opportunities for securing non-volatile memories. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 610–614.

[5] A. Awad, Y. Wang, D. Shands, and Y. Solihin. 2017. ObfusMem: A low-overhead access obfuscation for trusted memories. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 107–119. https://doi.org/10.1145/3079856.3080230

[6] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. A. Zubair. 2019. Triad-NVM: Persistency for Integrity-Protected and Encrypted Non-Volatile Memories. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 104–115.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[8] Chenyu Yan, D. Englender, M. Prvulovic, B. Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *33rd International Symposium on Computer Architecture (ISCA'06)*. 179–190. https://doi.org/10.1109/ISCA.2006.22

[9] Siddhartha Chhabra and Yan Solihin. 2011. I-NVMM: A Secure Non-Volatile Main Memory System with Incremental Encryption. *SIGARCH Comput. Archit. News* 39, 3 (June 2011), 177–188. https://doi.org/10.1145/2024723.2000086

[10] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. 2020. Persist Level Parallelism: Streamlining Integrity Tree Updates for Secure Persistent Memory. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 14–27.

[11] Intel. 2020. Build Persistent Memory Applications with Reliability Availability and Serviceability. [Online; accessed 7-March-2021].

[12] Intel. 2020. The Challenge of Keeping Up with Data. [Online; accessed 7-March-2021].

[13] Intel. 2020. Deprecating the PCOMMIT Instruction. https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html [Online; accessed 7-March-2021].

[14] S. Liu, A. Kolli, J. Ren, and S. Khan. 2018. Crash Consistency in Encrypted Non-volatile Main Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 310–323. https://doi.org/10.1109/HPCA.2018.00035

[15] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing Memory and Storage Support for Non-Volatile Memory Systems. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/3307650.3322206

[16] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 135–148. https://doi.org/10.1145/3093337.3037730

[17] PMDK. 2020. Persistent Memory Programming. "https://pmem.io/pmdk/". [Online; accessed 7-March-2021].

[18] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. 2007. 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)*. 183–196. https://doi.org/10.1109/MICRO.2007.16

[19] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi. 2018. Morphable Counters: Enabling Compact Integrity Trees For Low-Overhead Secure Memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 416–427. https://doi.org/10.1109/MICRO.2018.00041

[20] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi. 2018. SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 454–465. https://doi.org/10.1109/HPCA.2018.00046

[21] S. Shin, J. Tuck, and Y. Solihin. 2017. Hiding the long latency of persist barriers using speculative execution. In *44th Annual International Symposium on Computer Architecture (ISCA 2017)*. 175–86. https://doi.org/10.1145/3079856.3080240

[22] Bo Yang, Sambit Mishra, and R. Karri. 2005. A High Speed Architecture for Galois/Counter Mode of Operation (GCM). *IACR Cryptol. ePrint Arch.* 2005 (2005), 146.

[23] M. Ye, C. Hughes, and A. Awad. 2018. Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 403–415. https://doi.org/10.1109/MICRO.2018.00040

[24] Vinson Young, Prashant Nair, and Moinuddin Qureshi. 2015. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. *ACM SIGPLAN Notices* 50 (05 2015), 33–44. https://doi.org/10.1145/2775054.2694387

[25] J. Zhou, A. Awad, and J. Wang. 2020. Lelantus: Fine-Granularity Copy-On-Write Operations for Secure Non-Volatile Memories. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 597–609. https://doi.org/10.1109/ISCA45697.2020.00056

[26] K. A. Zubair and A. Awad. 2019. Anubis: Ultra-Low Overhead and Recovery Time for Secure Non-Volatile Memories. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 157–168.

[27] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo. 2018. 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)*. 442–454. https://doi.org/10.1109/MICRO.2018.00043