Anubis*: Ultra-Low Overhead and Recovery Time for Secure Non-Volatile Memories

Kazi Abu Zubair
University of Central Florida
Orlando, Florida, USA
kzubair@knights.ucf.edu

Amro Awad
University of Central Florida
Orlando, Florida, USA
amro.awad@ucf.edu

ABSTRACT

Implementing secure Non-Volatile Memories (NVMs) is challenging, mainly due to the necessity to persist security metadata along with data. Unlike conventional secure memories, NVM-equipped systems are expected to recover data after crashes and hence security metadata must be recoverable as well. While prior work explored recovery of encryption counters, fewer efforts have been focused on recovering integrity-protected systems. In particular, how to recover Merkle Tree. We observe two major challenges for this. First, recovering parallelizable integrity trees, e.g., Intel's SGX trees, requires very special handling due to inter-level dependency. Second, the recovery time of practical NVM sizes (terabytes are expected) would take hours. Most data centers, cloud systems, intermittent-power devices and even personal computers, are anticipated to recover almost instantly after power restoration. In fact, this is one of the major promises of NVMs.

In this paper, we propose *Anubis*, a novel hardware-only solution that speeds up recovery time by almost 10^7 times (from 8 hours to only 0.03 seconds). Moreover, we propose a novel and elegant way to recover inter-level dependent trees, as in Intel's SGX. Most importantly, while ensuring recoverability of one of the most challenging integrity-protection schemes among others, *Anubis* incurs performance overhead that is only 2% higher than the state-of-the-art scheme, Osiris, which takes hours to recover systems with general Merkle Tree and fails to recover SGX-style trees.

CCS CONCEPTS

• **Security and privacy** → **Cryptography**; **Systems security**; *Database and storage security*; • **Computer systems organization** → **Architectures**; • **Hardware** → *Emerging technologies.*

KEYWORDS

Security, Non-Volatile Memories, Persistence, Persistent Security

ACM Reference Format:
Kazi Abu Zubair and Amro Awad. 2019. Anubis*: Ultra-Low Overhead and Recovery Time for Secure Non-Volatile Memories. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307650.3322252

1 INTRODUCTION

Emerging Non-Volatile Memories (NVMs), such as Phase-Change Memory (PCM), are considered promising contenders for replacing DRAM due to their scalability [1, 2], non-volatility, ultra-low idle power and low latency[3–5]. On the other hand, emerging NVMs bring in a set of new challenges for memory management and security; since NVMs retain data even after power loss, necessary measures for security and persistence of security metadata must be taken throughout the lifetime of the system and across system reboots/crashes [6–8]. Security and persistence have to be considered during the design time of NVM-based memory systems. In fact, the persistence of data has to be accompanied by persisting related security metadata to ensure secure and functional recovery [7, 8].

State-of-the-art secure memory systems employ counter-mode encryption[7–9]. In counter-mode encryption, each memory block (cacheline) is associated with a specific counter that is used along with processor key to encrypt/decrypt the data block once written/read to/from memory. While ensuring the confidentiality of such counters is considered unnecessary, their integrity must be protected; replaying an old counter would compromise the security of counter-mode encryption. To protect the integrity of counters, Merkle Tree is generally used. Merkle Tree consists of hashes over hashes where the leaves are the counters, and finally the root of the tree is always kept in the processor chip. Each counter update changes such root value, and hence any tampering will be detected due to root mismatch. Integrity trees have been also deployed in commercial products for secure processors, e.g., Intel's SGX. Moreover, integrity trees have been considered thus far the most secure scheme with low on-chip storage overhead; only the root needs to be kept inside the processor chip.

Implementing integrity trees with NVM memory systems is particularly challenging; updates to counters result in updating all levels up to the root. Thus, to ensure consistency after a crash, such updates need to occur atomically. Second, since NVMs have a slow, power-consuming and limited number of writes, updating all levels (can be tens of levels) of Merkle Tree on each counter update is impractical. Third, in some integrity trees, such as SGX-style, which offers parallel updates, recovering the tree by relying on recovering leaves (counters) is infeasible. Finally, reconstructing Merkle Tree is required to verify the first access to memory, however, for terabytes

[*] Anubis is an Egyptian god known for embalming and watching over the dead. Our work is mainly inspired by Anubis's capabilities to embalm and track the lost metadata blocks.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322252

of memory such a recovery process can take many hours. Practical NVM capacities are expected to reach terabytes per processor socket. For instance, Intel's Xeon server is expected to be equipped with 6TB NVM memory[10].

In this paper, we aim to provide recoverability and ensure ultra-low recovery time while incurring minimal run-time overheads. Recoverability is perhaps the most promising system feature that NVMs provide. Thus, adding security primitives such as encryption and integrity-protection should also consider recoverability. Moreover, emerging NVMs are promising for data centers, cloud systems and HPC systems, mainly due to their ultra-low idle power. In such systems, availability is a strict requirement. For instance, to meet the well-known availability target of 99.999% (five nines rule), the system can be down for a maximum of 5.25 minutes per year. Our results show that recovering a practical NVM size (8TB) with current secure processor implementations would take 7.8 hours, which is considered unacceptable. For instance, for each minute of the system being down in Amazon's cloud system, it is estimated to cost 70 thousand dollars per minute[11]. A simple example would be an in-memory database system, where a crash occurs right after a transaction is committed. In such a case, the whole Merkle Tree must be recovered first to be able to verify integrity before completing any new transactions or enquiries.

The main source of the security metadata crash inconsistency problem is the not-persisted-yet updates to Merkle Tree and encryption counters. In conventional systems, most of these updates occur in volatile caches inside the processor chip, however, without strictly persisting them in memory. Thus, once a crash occurs, the data might be written to memory while its updated encryption counter (and updated Merkle Tree nodes) has not been reflected in memory yet. Therefore, after recovery, the system will have an inconsistency between the versions of data and its corresponding security metadata. On the other hand, strictly persisting updates to such security metadata would eliminate such inconsistency, however, at the cost of tens of additional memory writes for each normal write. Given the limited write-endurance and very slow NVM writes, such solution (strict persistence) is considered impractical. Meanwhile, since counter and Merkle Tree caches can hold hundreds of thousands of security metadata cache blocks[8, 12], guaranteeing enough power to ensure flushing their content is impractical. In fact, state-of-the art processors have a limited number of entries (only tens of write entries) that are guaranteed to be persisted by the system. For instance, in recent Intel's processors, a small buffer (tens of entries) co-located with memory controller is called *Write Pending Queue (WPQ)* [13, 14]. Write operations are considered persistent once they reach the WPQ buffer. Flushing WPQ to NVM in case of power loss mainly relies on a system feature called *Asynchronous DRAM Self-Refresh (ADR)* [13, 14] which guarantees enough power to flush the contents of WPQ. Thus, given the limited ability of ADR, high costs of uninterruptible power supplies, the demand for battery-free solutions, area and environmental constraints, it is important to innovate new mechanisms to ensure persistence of security metadata [7, 8].

The state-of-the-art solution, Osiris[7], enables recovery of encryption counters (the leaves of Merkle Tree) through leveraging data ECC bits to identify the most recent counter value used for encryption. While Osiris relies on Merkle Tree for final verification of the candidate counter values, it overlooks the recoverability of Merkle Tree. However, similar to any other counter recovery scheme, Osiris would take hours to recover the system after a crash. Before completing the first memory access, counter recovery schemes need to fix the counters and then build up the whole Merkle Tree upon all the counters (leaves), and hence iterating over a huge number of memory blocks. Moreover, Osiris does not consider the case where recovering encryption counters alone is insufficient to reconstruct some Merkle Trees as in commercial secure processors. Another work, selective persistence [8], relaxes atomic persistence of encryption counters for non-persistent data through software modification and API. However, selective persistence can lead to certain security vulnerabilities [7], its overhead scales with the amount of persistent data, and fails to ensure recoverability of parallelizable trees and also incurs significant overheads for reconstructing Merkle Tree. In both studies, Osiris [7] and selective persistence[8], Merkle Tree reconstruction has been overlooked and neither its recoverability nor its recovery time with practical capacities were considered or discussed.

To overcome such major challenges, and enable implementing security primitives effectively with persistent memory systems, we propose *Anubis*. *Anubis* is a novel hardware-only solution that enables ultra-low recovery time and additionally ensures recoverability of parallelizable trees (as in SGX). *Anubis* is based on a key observation: **persistently** tracking the addresses of blocks in counter and Merkle Tree caches can be used to significantly reduce recovery time. However, the addresses in such caches do not change as frequent as content; addresses only change when a miss occurs, and hence the overhead to track them in memory is minimal. By tracking such addresses, after a crash, the system needs to only rebuild the affected parts of the tree. Moreover, for SGX-style trees, we shadow the metadata cache and elegantly protect the integrity of such shadow version. Thus, at a recovery time of SGX-style tree, we only need to cache back such shadow version. To the best of our knowledge, our paper is the first to address the recovery time of general trees and the recoverability of SGX-style trees. Our main solution is inspired by *Anubis* (the ancient Egyptian god) for the ability to watch over the dead (by tracking potentially lost metadata) and embalming (by keeping an integrity-protected shadow copy of the cache in persistent memory).

To evaluate the performance overhead of our proposed solution, we use Gem5 [15], a cycle-level simulator to run memory-intensive benchmarks from SPEC2006 suite[16]. Our evaluation shows that, on average, *Anubis* reduces the performance overhead from **63%** (for strict-persistence) to only **3.4%**, which is nearly as low as Osiris overhead (**1.4%**). Most importantly, *Anubis* achieves a recovery time of **0.03s**, whereas Osiris requires an average of **7.8 hours** for 8TB to recover both encryption counters and Merkle Tree. Besides, *Anubis*'s recovery time is only a function of the security metadata cache size and does not increase linearly with memory size as in other schemes. In summary, our contributions in this paper are the following:

- We propose *Anubis*, a novel memory controller design that enables low-overhead and low recovery time for integrity-protected systems.

- The first ever solution to enable recovery of parallelizable Merkle Trees such as Intel's SGX counter tree.
- *Anubis* achieves recovery time speedup (10^7 speedup) that keeps secure memory practical when used with persistent memory.
- To the best of our knowledge, this is the first paper to discuss the impact of Merkle Tree cache update scheme and Merkle Tree implementation (parallelizable vs. unparallelizable) on recoverability.

With all the above contributions, we believe that this paper will open up a new research direction that considers recoverability when implementing secure persistent memories.

The rest of the paper is organized as following. First, in Section 2, we present an overview of memory encryption schemes, crash consistency and atomicity of security metadata persistence. Section 3 presents quantitative results that demonstrate the problem. Later, in Section 4, we discuss the design of *Anubis* and its different variants. Section 5 presents our evaluation methodology. Later, Section 6 presents our evaluation for the *Anubis*. Section 7 discusses the related work. Finally, Section 8 concludes our paper.

2 BACKGROUND

In this section, we discuss the main aspects of secure NVM systems and related concepts.

2.1 Threat Model

Similar to the state-of-the-art approaches [6–9, 12], our threat model only trusts the processor chip. An attacker can possibly scan the memory, snoop the memory bus, replay memory packets, and can tamper with memory contents or memory bus packets. Attacks such as access pattern leakage, memory timing, power analysis, speculative execution and electromagnetic (EM) side channel attacks are beyond the scope of this paper.

2.2 Counter-Mode Encryption

In state-of-the-art secure processors, counter-mode memory encryption is used. Counter-mode encryption enables overlapping the encryption/decryption latency with fetching data by using Initialization Vectors (IVs) to generate encryption/decryption pads that will be merely XOR'ed with ciphertext/plaintext to complete decryption/encryption. Thus, fetching the data can occur concurrently with generating the pad, typically called One-Time Pad (OTP), from the IV. Additionally, by ensuring spatial and temporal uniqueness of such OTPs, the same data is very unlikely to generate the same ciphertext; part of the IV (counter) will change each time it is used for encryption, and the address of the block is used as a part of the IV to ensure spatial variation.

Since the read/write granularity in memory system is cache blocks, typically 64B, it is common to associate each cache block with a counter that will be used to establish the IV to be used for its encryption/decryption. However, counter-mode encryption strictly prohibits reusing the same IV with the same key as it enables known-plaintext attacks. Therefore, such counters should be large enough to reduce the overflow rate in which the whole memory must be re-encrypted with a new key. Unfortunately, such limitation imposes large storage overheads. Therefore, state-of-the-art schemes organize counters into major and minor counters (called split-counter scheme) [9], where a small minor counter, typically 7 bits, is associated with each cache block (64B), and a larger major counter, typically 64 bits, is shared across all cache blocks of the same page (4KB). By doing so, only when a minor counter overflows, the major counter is incremented and all the page will be re-encrypted with the new major counter.

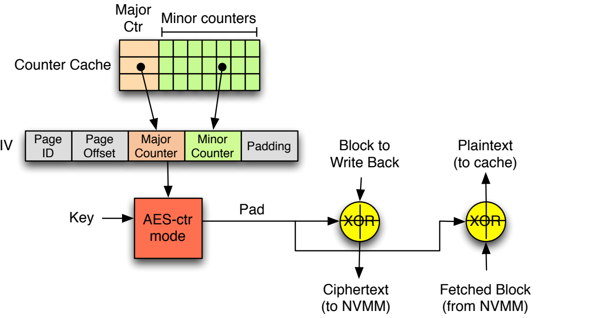


Figure 1: Counter-Mode Encryption in State-Of-The-Art Secure Memories [7, 12].

Figure 1 depicts how Initialization Vectors (IVs) are established from counter blocks. Note that page counters are packed in 64B counter blocks that contain 64 minor counters (one for each cache block in the page) and a major counter. However, SGX uses 56bit counters and packs 8 counters per 64B counter block. For the rest of the paper, we use split-counter scheme for the general memory encryption scheme and SGX-like 56-bit counters for SGX-style scheme.

2.3 Integrity Verification

Ensuring the integrity of counters is a major security requirement for the counter-mode encryption scheme. Typically, a root hash value that is calculated over all the counters is stored in the processor chip. However, to facilitate fast verification of counters and updating the root, a tree of hashes is typically used. By doing so, verifying a counter or updating its hash value would only require accessing the corresponding part of the tree instead of all other counters in the system. Such tree-of-hashes is typically referred to as *Merkle Tree*.

State-of-the-art implementations incorporate data integrity verification with counter integrity verification [9]. In particular, ensuring the integrity of the counter through Merkle Tree while protecting data through a MAC value calculated over the data and the corresponding counter would be sufficient from a security standpoint. Such systems that employ Merkle Tree for counters + MAC (over counter and data) are called *Bonsai Merkle Trees* [9].

In general, there are two types of Merkle Trees, non-parallelizable and parallelizable (similar to SGX-style) as will be discussed next.

2.3.1 Non-Parallelizable Merkle Tree. Figure 2 depicts a general Merkle Tree scheme that builds on the hashes over hashes. In this scheme, a number of counter blocks are hashed together using a cryptographic hash function stored in the processor. These hashes are hashed over and over again and a tree structure is formed with the 64B counter blocks being the leaves and one 64B hash at the

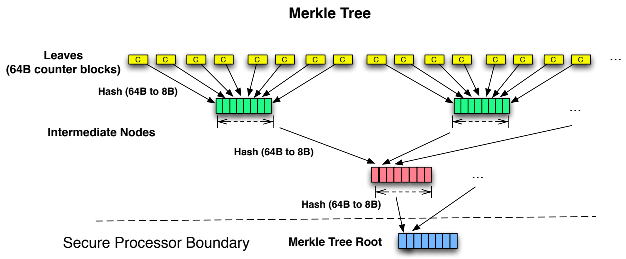


Figure 2: General Non-Parallelizable Merkle-Tree.

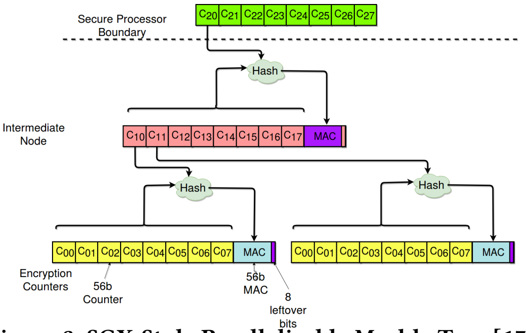

Secure Processor
Boundary
C20 C21 C22 C23 C24 C25 C26 C27
Hash
Intermediate
Node
C10 C11 C12 C13 C14 C15 C16 C17 MAC
Hash
Hash
C00 C01 C02 C03 C04 C05 C06 C07 MAC
C00 C01 C02 C03 C04 C05 C06 C07 MAC
Encryption
Counters
56b
Counter
56b
MAC
8
leftover
bits

Figure 3: SGX-Style Parallelizable Merkle Tree [17].

top of the tree as the root which is always kept inside the processor. As mentioned earlier, the counter blocks are organized using the split-counter scheme and we use Bonsai Merkle Tree style.

In this scheme, each level of the tree consists of hash values calculated over a number of its direct children nodes (those on the lower level). For instance, in each level, each node is 64B that consists of eight 8B hash values. Each 8B hash value has been calculated over a 64B node of the lower level, thus the tree is called 8-ary tree (each of the 8 nodes will have one parent node in the upper level). At the time of verification for a counter block, we need to verify the hash values of all its parents in upper levels until the first level hit in the cache as it has been already verified when brought to the processor chip. In the worst case, when all corresponding nodes in upper levels are missing, we need to verify each level that it matches the MAC calculated in its upper level up to the root. However, such verification steps can occur in parallel for each level. In contrast, updating counters would need updating upper levels sequentially; it is not feasible to compute the hash value of the next level before the current level. Thus, parallelizing updates to Merkle Tree levels is not achievable in this scheme.

One thing that is important to note here is that the whole Merkle Tree, including the root, can be reconstructed from the leaves (counter blocks). Thus, while the reconstruction time can be too long, just persisting counter blocks is sufficient to reconstruct the tree starting from the leaves all the way up to the root.

2.3.2 Parallelizable Merkle Tree. To enable parallelism, parallelizable Merkle Tree uses nonces for each tree node. A MAC is stored in each block with counters/nonces. The MAC is calculated over the nonces in the block and one nonce from the parent block using a secret hash function stored in the chip as shown in Figure 3. Whenever an encryption counter is incremented, the respective nonce in the parent nodes is incremented too. This way, a parallel update of MAC values is possible since each block can be updated separately by incrementing their nonces and calculating the MAC over the new nonces.

There are two design options for the update of such tree structure. As discussed in [18, 19], once an encryption counter is updated, it is only sufficient to update the upper nodes up to the first cache hit. This scheme does not immediately update the root. Once that node in the cache (one which is the most updated one in the cache and beyond which writes did not propagate last time) gets evicted, the parent node is incremented. Thus writes are propagated upward lazily relying on the assumption that such updates will be eventually propagated once evicted from the cache. The other design option is to always propagate the tree update up to the root regardless of the availability of intermediate nodes in the cache. In our design for such tree structure, we have adopted the former method similar to Synergy [20] and Vault [18].

Although such parallelizable tree structure allows parallel authentication as well as update of the tree, the recovery of the tree structure after a crash is challenging since the nonces are lost along with the hash stored in the same block. Unlike the general hash tree structure, it is impossible to regenerate the tree only from the encryption counters and hence impossible to verify counters' integrity if intermediate nodes are lost.

2.4 Recovery of Counters

Recently, Ye et al. [7] proposed a scheme to recover encryption counters after a crash. The authors discussed several ways including extending the data bus to include a portion of the counter, *phase*, which stores the low significant bits of the counter to relax the need to persist the counter block on each update. Furthermore, the authors propose a novel scheme, called Osiris, which leverages the encrypted ECC bits co-located with data as a sanity-check to help in retrieving the counter used for completing the encryption of data. By combining this with a stop-loss mechanism, persisting the counter every Nth update, only a few trials are needed to recover each counter through checking ECC bits sanity. For more details about the reliability and implementation of Osiris, we refer the readers to the original paper[7].

Our paper focuses on speeding up the recovery of Merkle Tree. Thus, restoring encryption counters is necessary but orthogonal to this work. While we will use Osiris to recover leaves (in case of general Trees), any other counter recovery schemes, e.g., extending data bus or writing phase bits along with data by adding extra burst, can work with our scheme. It is very important to note that our paper goal is to speed up recovery time by reducing the number of counters that needs to be recovered instead of how they are recovered. Moreover, we aim at enabling recovery in schemes where just recovering counters is insufficient to rebuild the tree.

2.5 Merkle Tree Reconstruction

Once all of the counters are corrected using Osiris, or other schemes, the Merkle Tree can be regenerated and the integrity of the counters can be verified by matching the root with the root of the tree stored securely inside the chip. However, rebuilding the Merkle Tree is required to verify the first access; verifying each level would require ensuring that all its children are up-to-date, and hence a recursive operation of calculating MACs over all the tree will be needed after recovery. Such a recursive process would lead to reconstructing the Merkle Tree. Unfortunately, since NVMs are expected to have huge capacities, such trees will be huge and would take hours to be reconstructed.

2.6 Relationship Between Recoverability and Tree Update Scheme

As discussed earlier, Merkle tree caches can be eagerly updated all the way up to the root on each write access, or updated lazily. In eager update scheme, the root always reflects the most recent state of the tree and thus can be used for verification after recovering the system. In contrast, in the lazy update scheme, the Merkle Tree and counter caches have the most recent values, but the root might be still stale. Considering persistent memory systems, lazy update scheme can be used if and only if there is a way to completely recover the cached content and verify their integrity (and freshness) using means different than relying on the root; the root is stale and no longer can be used as a way to verify the most-recent content. However, for the eager update scheme, the root can be used to verify the integrity of all contents, including the recovered tree updates.

SGX-style trees are difficult to recover by only knowing the root value; the updated intermediate nodes must be also recovered and used for verification of upper and lower levels due to inter-level dependency. Thus, for SGX-style trees, the most suitable scheme to ensure recovery would be a secure and verifiable recovery of Merkle tree cache, while using a lazy update scheme. In contrast, for Bonsai Merkle tree, we can rely on either lazy or eager update schemes, however, eager update scheme would incur much less run-time overheads since it does not require frequent memory updates to enable verification and restoration of the exact state of the Merkle Tree cache and counter cache; just recalling which addresses could have been potentially lost, and fix them, is sufficient.

In summary, if during a crash, the root does not reflect the most recent value, we need a mechanism to securely recover and verify the updates in Merkle Tree and counter caches, i.e., guaranteeing that we know the values of all updates in the cache and verify their freshness. However, if the root is strictly updated, i.e., using eager update scheme, then we can recover the tree and counters, and verify them using the root. The only exception is that for SGX-style trees, the eager scheme is insufficient due to the reliance on inter-level verification (See Figure 3).

2.7 Atomicity of Data and Security Metadata Updates

In modern processor systems, there is a feature called ADR that allows enough power to flush the contents of the *Write Pending Queue (WPQ)* inside the processor to the NVM memory [13, 14] once a crash occurs. However, the WPQ size is limited tens of entries. Since anything gets inserted in WPQ is guaranteed to persist, i.e., in the persistent domain, we have to ensure that the content of WPQ ensures consistency or at least there is a way to bring the system to a consistent state after power restoration. This leaves us with a challenge on how to atomically insert updates to data, counter, Merkle Tree nodes and *Anubis* updates in WPQ. To solve this problem, we rely on internal *persistent registers* to hold all the updates (data, counter, Merkle Tree nodes and *Anubis* updates) before attempting to insert them individually into the WPQ. Note that once the data arrives the memory controller, it is placed in volatile write buffer (outside the persistent domain) and the entry is considered persisted only after its accompanying updates (counter and Merkle Tree nodes) and its data are placed in the *persistent registers*. Later, the contents of the persistent registers are individually inserted in the WPQ. If any crash occurs while we are copying the content from persistent registers to WPQ, then the memory controller will try again to insert these updates to WPQ after recovery. Note that if a crash occurs while we are still trying to make the updates on persistent registers, then such write is already lost as it has never reached the persistent domain. We can use *DONE_BIT* to find out if the crash occurred while there is a valid content in persistent registers or not; DONE_BIT is only set after all affected values have been written to persistent registers and cleared after all persistent registers have been copied to WPQ. Note that our scheme is similar in spirit to REDO logging, however, using internal persistent registers. Note that using internal persistent registers have been also assumed in state-of-the-art work [7]. Figure 4 depicts the design of atomic updates of Merkle Tree in strict persistence scheme.

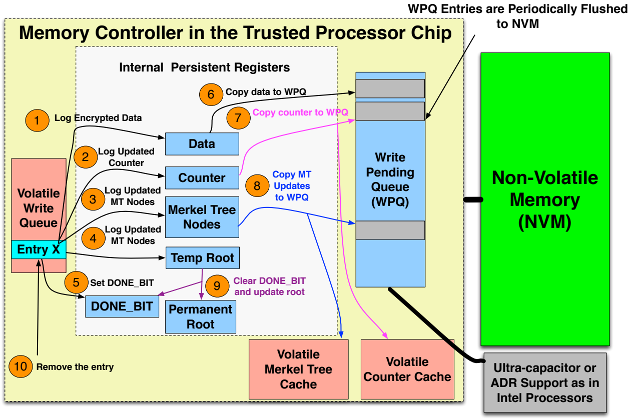


Figure 4: Atomic Persistence of Integrity-Protected NVMs.

For the rest of the paper, we use a two-stage commit process that leverages persistent registers to implement the discussed REDO logging scheme. However, to avoid slow writes and limited endurance of NVM registers, such persistent registers are implemented as volatile registers that are backed by (copied once crash detected) NVM registers, or simply dedicating few entries from the WPQ as volatile registers and leverage ADR to copy them to internal NVM registers. Note that multiple write entries in WPQ can be flushed concurrently to exploit the bank-level parallelism in NVM and avoid serializing writes to NVM.

3 MOTIVATION

As discussed earlier, parallelizable Merkle Trees, as in SGX-style trees, are very challenging to recover. Unlike typical Merkle Tree, the verification of each level counters also relies on the cryptographic hash values stored in their children nodes; the hash value in their children levels are calculated over the counters in that child node and a counter in the upper level, i.e., each upper-level counter is used as a version number that will be verified by the children hash value and later verified by the hash value of its neighboring counters along with its parent counter. The process continues until all corresponding parent counters are verified, however, if such parent counters exist in the cache, it means that they are already verified and thus it is sufficient to stop once the lowest level hit in the cache is found. Unfortunately, if any of such intermediate nodes gets lost during a crash, even if the root counters are saved inside the processor, it is impossible to verify the integrity of leaves; they strictly rely on verifying all their parent counters. Thus, due to this inter-level dependence in this style of Merkle Tree, it needs special handling to enable their recovery. Meanwhile, general Merkle Tree implementations, such as Bonsai Merkle Tree, can be completely reconstructed if we can recover encryption counters (the leaves) as explained in prior work [7]; upper levels of Merkle Tree are simply the hash values of lower-levels, and thus as long as the root hash value matches after reconstruction, the counters are considered verified. However, even for general Merkle Tree, the recovery time is impractical for practical NVM capacities, e.g., 4TB or 8TB[10]. Figure 5[1] shows the recovery time for different NVM capacities assuming encryption counters can be recovered using state-of-the-art counter recovery scheme [7].

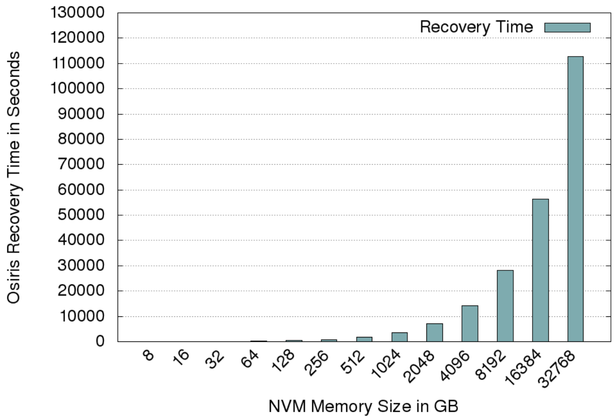


Figure 5: Recovery Time for Different Memory Sizes (Using Osiris).

Counter recovery schemes rely on reading data blocks to use their accompanied ECC bits as a sanity-check for recovering the used encryption counter [7]. Thus, the recovery time scales linearly with the size of the memory (number of data blocks). For instance, we can see that at 8TB NVM capacity, almost 8 hours are needed to recover the system. With the expected huge capacities of NVMs, any operation that is $O(n)$, where n is the number of data blocks (typically 64B) in memory, can take impractical time and should be avoided, as also observed by Swift et al.[21].

Based on these two key observations, we follow the following two main design principles:

- Enable $O(n\_cache \times log_{TreeArity}(n))$ recovery time, where n_cache is the number of blocks in counter cache, instead of $O(n)$. For example, $O(1000 \times log_8(n))$ instead of $O(n)$. However, while also ensuring minimal run-time overhead. By doing so, for 8TB memory, instead of iterating over $\sim 13 \times 10^{10}$ blocks, we only need $\sim 1000 \times 12$ blocks, which is more than 10^7 speed up in recovery time.
- For SGX-Style Merkle Tree, it is insufficient to recover encryption counters and having the most recent root value. There must be a mechanism to reliably recover and verify the integrity of all lost intermediate nodes.

Without such two major design principles, the recovery of integrity-protected NVM memory systems would be either impractical (due to significant recovery time) and/or completely infeasible due to lost intermediate nodes in inter-level dependent trees. Thus, we propose a new memory controller design, *Anubis*, that realizes both design principles.

4 ANUBIS DESIGN

In this section, we describe the design options and details for *Anubis*.

4.1 Tracking Updated Security Metadata

One key observation we have is that it is sufficient to *persistently* track the addresses of the blocks in the Merkle Tree and counter caches to significantly reduce recovery time; only the blocks of the tracked addresses have been possibly updated without being persisted. Thus, by having the ability to identify the addresses of the counter blocks that were in the cache at the time of the crash, we only need to iterate through their corresponding counter blocks. Similarly, by tracking the addresses of the Merkle Tree nodes in the Merkle Tree cache, after fixing lost counters (using Osiris [7]), reconstructing general Merkle Tree can be implemented by starting from leaves, fixing those identified as lost, then going to the upper level and fix those identified as lost, continuously until reaching the top level. The order of fixing is important; repairing upper levels relies on fixing lower levels first.

In its simplest form, *shadow-tracking* can be implemented by reserving the cache size in NVM, e.g., a 128KB will be reserved in NVM for shadowing the addresses in the 128KB counter cache. During counter cache miss event, based on the location of the victim block in the data array of the counter cache, the address of the new block will be written to NVM on the offset corresponds to the location in the cache, as explained in Figure 6. Similar approach can be used for shadow-tracking Merkle Tree.

Note that the position of the block in the counter cache remains fixed for its lifetime in the cache; LRU bits are typically stored and changed in the tag array. Since the miss rate of counter cache is typically very small, the additional writes will be minimal.

As mentioned earlier, updated nodes in both, Merkle Tree and counter caches, must be tracked. For terminology, we refer the counter shadow-tracker as *Shadow Counter Table (SCT)*, whereas the Merkle Tree shadow-tracker is called *Shadow Merkle-tree Table(SMT)*. In both cases, the storage overhead is minimal, e.g., for a 128KB counter cache size and 8TB memory, the overhead is only

[1] Due to the impracticality of simulating recovery time for large capacities, we calculate recovery time by counting the number of hash values and nodes need to be fetched and updated from memory and assume each would cost 100ns (fetch from memory, hash calculation and/or decryption, as assumed in [7]).

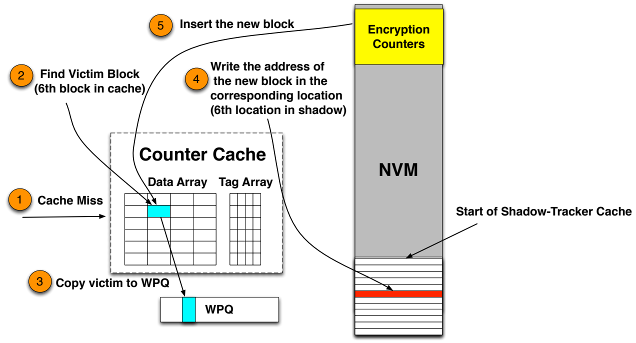

5 Insert the new block
Encryption Counters
2 Find Victim Block (6th block in cache)
4 Write the address of the new block in the corresponding location (6th location in shadow)
NVM
Counter Cache
Data Array
Tag Array
1 Cache Miss
Start of Shadow-Tracker Cache
3 Copy victim to WPQ
WPQ

Figure 6: Shadow Table

$\frac{128KB}{8TB}$. Figure 9(a) shows how SCT and SMT memory blocks are organized.

Anubis has two different schemes, one suitable for general Merkle Tree structure (e.g. Bonsai Merkle Tree) and another suitable for SGX-style tree. We term the first scheme as AGIT (Anubis for General Integrity Tree: Section 4.2) and the second one as ASIT (Anubis for SGX Integrity Tree: Section 4.3).

4.2 Anubis for General Integrity Tree (AGIT)

4.2.1 AGIT Read: Tracking Metadata Reads. As discussed earlier in Section 2.6, when eager cache update scheme is used, general Merkle Tree can be recovered by restoring the leaves and updating the tree upwardly before finally verifying that the resulting root matches that inside the processor. Thus, we can directly apply the idea of shadow-tracking for both Merkle Tree and counter cache to speed up the recovery process; only lost nodes and counters need to be fixed. For both caches, the shadow regions are updated on each cache miss, i.e., before reading a metadata block from memory, and hence we call it *AGIT Read* scheme. Note that such shadow regions are merely used for recovery acceleration; once recovered, as usual, the root will be used to verify all counters and Merkle Tree nodes as they are getting read into the processor chip. Thus, any tampering with the content of shadow regions or unaffected counters (were not in the cache) will lead to root mismatch when the affected (not recovered correctly or tampered with) is read into the processor chip.

Figure 8(a) illustrates the operation of AGIT Read. As shown in the figure, once a memory request arrives at the memory controller (step ①), the required encryption counter and Merkle Tree nodes are retrieved from memory (as shown in Steps ② and ③), in case not present in the cache. Before inserting any of the counter or Merkle Tree blocks in the shadow cache, *Anubis* prepares the address blocks (used for tracking) of the counter and Merkle Tree blocks and insert them into the WPQ(as shown in Step ④) before inserting them in the cache (in Step ⑤). Note that since the tracking blocks (prepared in Step ④) are already in WPQ (persistent write queue), they are considered persistent, however, they will be eventually written to the SCT/SMT region in memory as soon as WPQ is flushed or such entries get evicted from WPQ.

Although AGIT-Read scheme performs reasonably well with applications that are not read intensive, read-intensive applications may trigger extra writes and update the SCT and SMT very frequently.

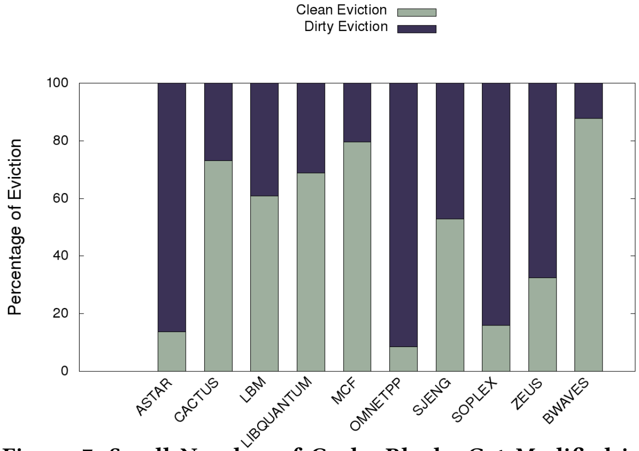
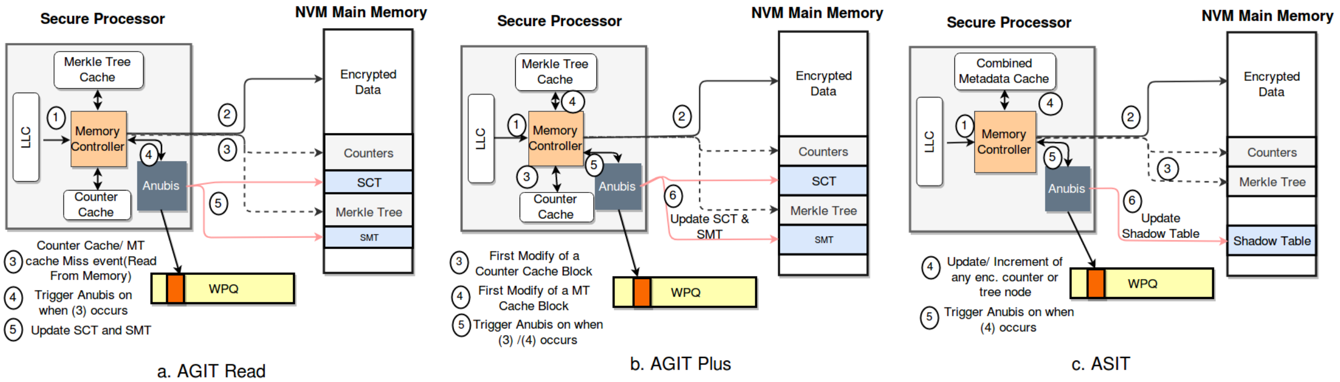


Figure 7: Small Number of Cache-Blocks Get Modified in Cache.

4.2.2 AGIT Plus: Tracking Metadata Modifications. In AGIT-Read, SCT and SMT are updated whenever some metadata are brought into the cache disregarding the fact that some metadata would never be modified in the cache. In fact, a significant number of blocks leave the cache without any modification. As illustrated in Figure 7, most applications evict a large number of cache-blocks from the counter cache that are clean. Hence, only tracking addresses of modified blocks provide the same recoverability but with reduced overhead. Moreover, most dirty cache-blocks are updated multiple times in the counter cache and Merkle Tree cache. In fact, Merkle Tree cache blocks reside in the cache and get modified more than counter cache blocks. Only tracking once during a dirty cache block's lifetime is sufficient to successfully recover the system.

Based on these observations, AGIT Plus reduces extra updates to the shadow tables by acting only whenever metadata is first modified in the Counter Cache or Merkle Tree Cache. This reduces the overhead of AGIT read significantly without hurting the recoverability. AGIT-Plus (as shown in Figure 8-b) is similar to AGIT-Read except that it triggers Anbuis only at the first update to a counter or Merkle Tree blocks in the cache (as in steps ③ and ④), i.e., setting the dirty bit for the first time. Before completing the update to caches, the generated shadow blocks must be inserted in WPQ.

4.2.3 AGIT Recovery Process. The recovery process of AGIT is straightforward. Once the system is booted up upon recovery, the system starts scanning the content of SCT to get the list of possibly lost updates in the cache. For each address, the data blocks (correspond to the possibly lost counters) are read and used to recover counter using Osiris as discussed earlier. Note that any other counter recovery scheme would likely have to read the data block, e.g., using phases or extending data bus. Later, once the affected counters are fixed, AGIT scans through SMT to search for possibly lost updates in the first level (immediate parents of counters) and recalculate the nodes' values based on their immediate children (counters). Later, once the 1st level is fixed, AGIT scans SMT for possibly lost updates to the second level and fix them by calculating


Secure Processor
NVM Main Memory
Merkle Tree Cache
LLC
Memory Controller
Anubis
Counter Cache
Encrypted Data
Counters
SCT
Merkle Tree
SMT
WPQ
1
2
3
4
5
3 Counter Cache/ MT cache Miss event(Read From Memory)
4 Trigger Anubis on when (3) occurs
5 Update SCT and SMT
6 Update SCT & SMT
3 First Modify of a Counter Cache Block
4 First Modify of a MT Cache Block
5 Trigger Anubis on when (3) /(4) occurs
Combined Metadata Cache
Shadow Table
6 Update Shadow Table
4 Update/ Increment of any enc. counter or tree node
5 Trigger Anubis on when (4) occurs

a. AGIT Read b. AGIT Plus c. ASIT

Figure 8: Anubis Operation.

Algorithm 1: AGIT Recovery

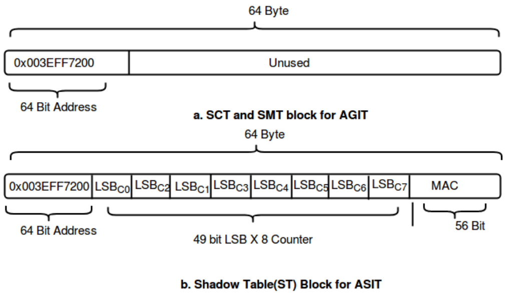
```
Read $SCT$ and $SMT$;
for $SCT_i$ in $SCT$ do
    Read Counter Block at address stored in $SCT_i$;
    for $Counter$ in $CounterBlock_i$ do
        Read $DataBlock_j$;
        Fix $Counter$ using Osiris;
    end
end
Classify SMT entries based on their level in tree ;
$MaxLevel \leftarrow$ Total Merkle Tree Levels;
$AffectedNodes_m \leftarrow$ Total Affected Nodes at Level m;
m $\leftarrow$ 0;
for $m \leq MaxLevel$ do
    for $Node$ in $AffectedNodes_m$ do
        Read all child nodes of $Node$;
        Create hash of child nodes and replace $Node$;
    end
    m $\leftarrow$ m +1 ;
end
if Stored root matches new root then
    System recovered;
else
    System is unrecoverable;
end
```

their values based on their immediate children (level 1). AGIT proceeds with the same process by going up in the tree level by level and eventually reach the root of the tree. Once at the root level, the resulting root from the calculated tree will be compared against what is in the processor chip to find out if the recovery has been successful. If the resulting root mismatches what is in the processor chip, then the recovery process has failed, and the system raises a warning about that. Note that the speed up in recovery mainly stems from the fact that we only need to fix the lost counters and Merkle Tree nodes than naively iterate through all the blocks as in systems without *Anubis*.

4.3 Anubis for SGX Integrity Tree (ASIT)

Unlike general trees, SGX-style integrity tree advocates for fast updates through limiting dependence between tree levels to only a counter on the upper level. Thus, on each update, affected nodes on different levels can be updated by calculating the MAC over their counters and the updated counter at the upper level [17]. However, this comes at the expense of extra complexity during reconstruction after a crash. Each intermediate node depends on a counter on the upper level, and counters of each level are verified using the MAC value co-located with each node. Thus, by losing the MAC values on different levels, it becomes infeasible to verify the integrity of the tree. Meanwhile, reproducing the MAC values of the intermediate tree is not safe until the counters of the level are verified.

In SGX, 8 of the 56-bit encryption counters are stored along with a 56 bit hash in one single cache line. Each parent node(Merkle Tree Node) also contains 8 counters (56 bits each), and a 56-bit hash value. However, as mentioned earlier, it is very challenging to recover the tree to its previous state after a crash and most of the time quite impossible if some intermediate nodes in the tree are missing. ASIT aims to provide a book-keeping mechanism that tracks the tree during run-time and recovers after a crash very quickly. Since encryption blocks in SGX have a similar structure to intermediate levels, a single *metadata cache* is typically used. For the shadow table, we also merge the SCT and SMT into one larger *Shadow Table (ST)* with a size similar to metadata cache. Figure 9(b) shows the organization of the Shadow Table for ASIT scheme.



a. SCT and SMT block for AGIT

b. Shadow Table(ST) Block for ASIT

Figure 9: Shadow Table Organization in AGIT and ASIT.

4.3.1 ASIT Metadata Tracking.

ASIT's main idea is inspired by *Anubis*'s embalming capability. In particular, ASIT aims to have an exact persistent copy of the content of metadata cache before the crash. However, such a shadow copy must have its integrity protected against any possible tampering. By doing so, it is sufficient to just restore the metadata cache by copying back the shadow cache after verifying its integrity.

In ASIT scheme (Figure 8-c), on each update to encryption counters (due to write requests) in the cache, the Shadow Table (ST) is updated with the modified cachelines in the cache. As described earlier (Section 2.6), an eager update scheme is inappropriate for

SGX-style trees; having a root value that reflects the most recent tree is insufficient to recover the tree. Meanwhile, strictly tracking all changes to counters and Merkle Tree would incur significant overheads with eager update scheme; each write would incur 12 writes to the shadow region. Therefore, we opt for using lazy update scheme while strictly tracking the changes to the metadata cache; in the lazy scheme only one block is typically updated on each memory write, and thus it is more practical to track such updates compared to eager update scheme. However, the cost is the need to fully protect the integrity of the shadow-table through a small general Merkle Tree (3-4 levels) with its root be persistent and never leaves the processor chip. The updates to the tree protecting the shadow table use eager cache update scheme, i.e., the root of the shadow table tree reflects the most recent state of the shadow table.

Each Shadow Table block contains the following elements (Figure 9-b):

- **Address:** 64-bit address of the Merkle tree block/encryption counter block modified in the combined metadata cache.
- **MAC:** 56-bit MAC value calculated over the updated counter values (nonces in that node).
- **Counter LSBs:** This part consumes most of the space in each Shadow Table entry and contains part of the LSBs of counters in that Merkle Tree node. 8 LSBs from each counter of the MT node is packed together into 49 bytes(49 bit each).

Whenever 49-bit LSBs of a counter overflows, the MT node is persisted so that the LSB value stored inside SMT can successfully recover the counter value. This ensures that the tree counters are recoverable using the MSB of the memory version of the counters, and the LSBs in the shadow block. Since 49-bit LSB overflows very rarely, the overhead of persistence due to overflows is negligible.

Protecting Shadow Table: As mentioned earlier, since, the original root can be stale and hence no longer can be used for verifying integrity, a small non-parallelizable Merkle tree structure is maintained just to provide integrity protection of the Shadow Table(ST). For 256kB Cache size, only a tree of four levels (8-ary) needs to be maintained. However, there is no need for persisting this tree in memory. It is sufficient to securely keep the root of such a tree, we call it (*SHADOW_TREE_ROOT*), as verification is done only during recovery and is very fast. It should be noted that, in AGIT scheme, such secondary tree to protect the shadow table is not necessary; if attacker omits or tampers with entries in shadow caches, then the resulting corruption in counters or Merkle Tree will be eventually detected due to root mismatch. To avoid potential deadlock scenario when evictions could occur due to insertion of blocks from the shadow region tree, we dedicate a small percentage of the metadata cache for the shadow region tree. Such part of the cache does not need to be shadowed.

4.3.2 *ASIT Recovery Process.*

The recovery process in the ASIT scheme is different than that of the AGIT scheme in the following two ways. First, Osiris (or any counter recovery scheme) is no longer needed and hence no need to try different counter values to finish recovery; the LSBs and MAC are replaced directly from the SMT block. Second, instead of rebuilding the Merkle Tree, ASIT only recovers the metadata cache to its pre-crash state.

The following steps are required for the recovery in the ASIT scheme. First, *Anubis* reads the Shadow Table (ST) from the memory into the cache and regenerates *SHADOW_TREE_ROOT*, the root of the general tree that is responsible for the integrity of the Shadow Table. Next, this root is compared with the securely stored version of it in the on-chip NVM register. Later, once the ST's integrity has been verified, recovery starts by iterating over each Shadow Table block that has been loaded in the cache. For each Shadow Table block, their non-persisted memory counterpart (stale node) is also read and the LSBs and MAC values of that non-persisted node are replaced with the LSBs and MAC stored in the Shadow Table, i.e., only MSBs of counters are used from the stale node. Later, for each recovered node, we verify that MSBs were not tampered with by verifying the MAC value with the result of applying hash over the counters of the node and the counter in the upper node (from the cache if it was recovered).

Algorithm 2: ASIT Recovery

```
Read $ST$;
Regenerate $SHADOW\_TREE\_ROOT$ and verify;
Recover Tree Nodes
for all $ST_i$ in $ST$ do
    $Stale\_Node_i$ ← Read node at address($ST_i$) and place in cache;
    $Recoverd\_Node_i$ ← Replace LSBs and MAC in $Stale\_Node_i$ from $ST_i$;
end
Verify Integrity
for all $Recovered\_Node_i$ in $Metadata\_Cache$ do
    $Verify\_Integrity(Recovered\_Node_i)$
    if $Integrity\_Not\_Verified(Recovered\_Node_i)$ then
        The system is unrecoverable;
    else
        Do Nothing;
    end
end
```

Once the recovery is done, every recovered tree node will have the dirty bit set to 1. This way, the updates lazily propagate to the memory due to natural eviction.

5 METHODOLOGY

To evaluate the performance overhead of *Anubis*, we use Gem5[15], a cycle-level simulator. As illustrated in the Talbe 1, we simulate a 4-core X86 processor. We also use 16GB PCM-based Main Memory with parameters modeled after [22], similar to related work[7, 8]. Analysis of different sizes of the counter cache and Merkle tree cache is presented in Section 6.3. We use 11 memory-intensive applications from SPEC 2006 benchmark suit [23] to stress our model. For each application, we fast forward to a representative region then simulate 500M instructions.

Table 1: Configuration of the Simulated System.

Processor	
CPU	4 cores, X86-64, Out-of-Order, 1.00GHz
L1 Cache	Private, 2 cycles, 23KB,2-Way
L2 Cache	Private, 20 Cycles, 512KB, 8-Way
L3 Cache	Shared, 32 Cycles, 8MB, 64-Way
Cacheline Size	64Byte
DDR-based PCM Main Memory	
Capacity	16GB
PCM Latencies	Read 60ns, Write 150ns
Encryption Parameters	
Counter Cache	256KB, 8-Way, 64B Block
Merkle Tree Cache	256KB, 16-Way, 64B Block
SCT in AGIT	256KB
SMT in AGIT	256KB
ST in ASIT	512KB

In all our experiments, we model all integrity-protection and encryption aspects including Merkle Tree cache, counter cache, and hash calculation latency.

6 EVALUATION

In this section, we evaluate our scheme based on the baseline system and several state-of-the-art schemes. We evaluate the performance of AGIT and ASIT scheme in separate results and compare their sensitivity and recovery time.

6.1 AGIT Performance

To evaluate the AGIT scheme, we model and compare five schemes as follows:

① **Write Back(Baseline):** Simple counter mode encryption with write back counter cache and general Bonsai style Merkle tree cache.

② **Strict Persistence:** This is the strict persistence scheme where we persist every modification of the counters, and Merkle tree nodes up to the root.

③ **Osiris:** Stop-loss counter update employed with the write-back scheme. Similar to Osiris, we use stop-loss limit 4.

④ **AGIT Read:** *Anubis* for General Integrity Tree(AGIT) scheme with write-back and stop-loss counter mode encryption. SCT and SMT are updated at every counter cache and Merkle tree cache miss.

⑤ **AGIT Plus:** This is our optimized AGIT scheme with write-back and stop-loss counter mode encryption. SCT and SMT are updated whenever the contents of the cache are modified for the first time.

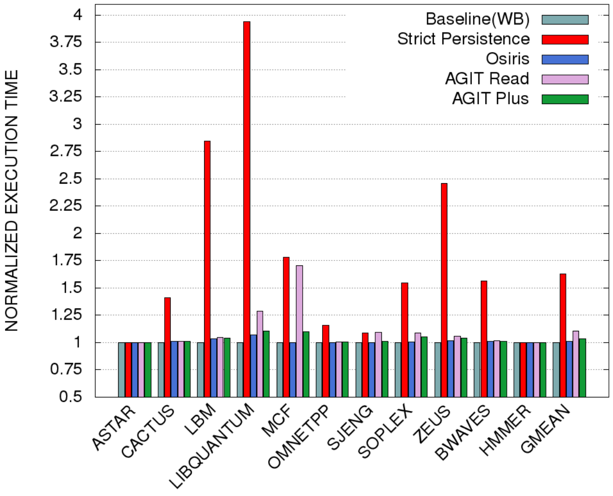


Figure 10: AGIT Performance.

Figure 10 shows the performance of AGIT scheme compared to other schemes. The Osiris-style stop-loss mechanism only adds a few more writes on top of the write-back baseline. This is because most of the counters easily get evicted before they reach the stop-loss limit. The low-overhead in Osiris scheme comes at the cost of impractical recovery time (as discussed in Section 3). The AGIT read scheme adds slightly more overhead on top of the Osiris scheme. However, its overhead is significantly high for applications that are read intensive (e.g. MCF). This is because MCF is read-intensive; few counters are actually written/updated in the cache before eviction (as shown earlier in Figure 7). This phenomenon can be noticed from the near identical performance of Write Back and Osiris for MCF which means the application is read intensive and evicts the counters long before they cross the stop-loss barrier. LBM is more write intensive than MCF and generates an insignificant number of read requests. Thus, AGIT read overhead is minimal in LBM but demonstrates slightly higher overhead in Osiris since many cachelines are written beyond the stop-loss limit. LIBQUANTUM performs both reads and writes more than the rest of the applications(except MCF for reads) and Osiris overhead is highest for LIBQUANTUM since it is the most write-intensive application we have tested.

AGIT Plus scheme is superior in terms of achieving both, low run-time overhead and practical recovery time. On average, AGIT plus only adds 3.4% extra overhead over the write-back scheme. The reason for the low overhead of AGIT-plus scheme is that it updates the shadow table in a relaxed way, which only occurs when a cacheline gets modified for the first time. Due to the varying write behavior of the applications, AGIT plus outperforms AGIT Read in case of read-intensive MCF. However, both schemes show negligible overheads, even for write-intensive applications like LIBQUANTUM, AGIT Read and AGIT Plus reduce the overhead by 9.17x and 24.5x, respectively. On average, AGIT Read incurs 10.4% overhead over write-back Scheme and reduces the overhead of the strict persistence by 5x. AGIT Plus performs even better, incurring only 3.4% overhead while reducing 17.4x overhead of strict persistence scheme.

6.2 ASIT Performance

For the ASIT scheme, we have modeled four schemes. Although we model Write Back and Osiris using SGX style tree, the only schemes that can recover such tree are Strict Persistence and ASIT:

① **Write Back:** Simple writeback scheme with SGX style Merkle Tree and counter mode encryptoin.

② **Strict Persistence:** Strict persistence scheme; every modification of the counters, and Merkle tree nodes up to the root are written back to the memory immediately.

③ **Osiris:** Similar to the Osiris scheme modelled in AGIT evaluation.

④ **ASIT:** This is the ASIT scheme working alongside write-back counter mode encryption.

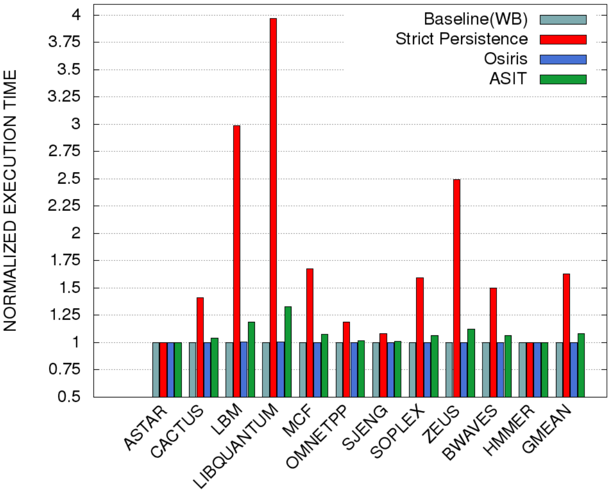


Figure 11: ASIT Performance.

Figure 11 illustrates the performance analysis of ASIT scheme. ASIT scheme, due to the nature of parallelizable trees, has to strictly track the meta-data writes in the cache, however, the average overhead is marginal when compared with the only scheme that ensures recoverability (strict persistence), 7.9% vs. 63%. The lazy write-back cache policy in ASIT scheme enables low overhead due to the reduced number of updates to cache, and hence reduced number of shadowing memory writes. Compared to the only scheme that supports recoverability (strict-persistence), even for write-intensive applications, LIBQUANTUM and LBM, the overheads are reduced approximately by 9x and 10x, respectively. The average overhead is only 7.9%, which makes ASIT the first practical scheme that enables recovery of SGX-style trees. In addition to the impractical performance overheads that the strict-persistence incurs, it causes at least an additional ten writes per memory write operation, which can significantly reduce the lifetime of NVMs. In contrast, ASIT only incurs one extra write operation per memory write.

6.3 Sensitivity Study

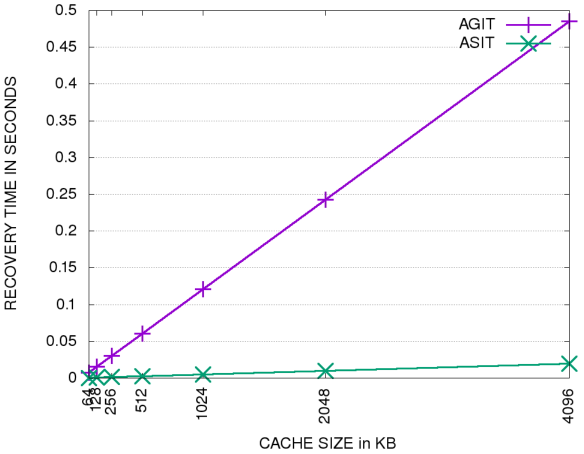


Figure 12: Recovery Time.

6.3.1 Recovery Time: Unlike previous models (e.g., Osiris), the recovery time in *Anubis* schemes is not a linear function of the memory size, i.e., the recovery time does not increase linearly with the increase of the memory size. In the AGIT and ASIT scheme, the recovery time is only the function of the cache size(Counter Cache, Merkle Tree Cache, or Combined Metadata-Cache) and the number of levels in the tree. Figure 12 shows how recovery time increases with the increase in cache size. Along the horizontal axis, both counter cache and Merkle tree cache sizes are increased by the same capacity. In comparison with the state-of-the-art *Osiris*, the recovery time for 8TB memory is ≈28193 seconds(≈7.8 Hours) as shown in Figure 5, while *Anubis* recovery time for extremely large cache sizes (4MB) is only ≈0.48s in AGIT, i.e., 58735x faster recovery time.

AGIT recovery time is almost more than double than the ASIT recovery time. This is because, in AGIT scheme, each counter block packs 64 counters(as in split counter scheme) each of which requires one encrypted block to be fetched from memory during recovery. However, SGX style counter cacheline (used in ASIT scheme) holds only 8 counters and ASIT recovery does not rely on the ECC bits to correct counters and hence requires only one read for the shadow block and one read for the affected node from memory during recovery. Besides, during recovery, ASIT scheme must bring one extra node from memory (if not in the cache) to read the upper counter during MAC generation. Generating hashes and MAC values in ASIT scheme consumes negligible compared to the read latency.

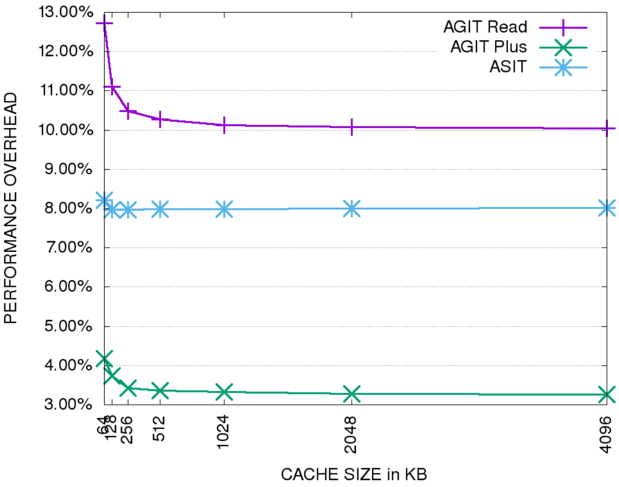


Figure 13: Sensitivity of Performance to Cache Size.

6.3.2 Performance sensitivity to Cache Size: The overall performance of each scheme improves with the cache size increase. As shown in Figure 13, the sensitivity of each of the scheme decreases after a certain level of cache size. Beyond 1MB, there is no significant improvement in performance with further cache size increase. It should also be noted that the least sensitive scheme with cache increase is ASIT. Although the write-back performance of ASIT scheme improves with an increase in cache size, the normalized performance improvement of ASIT scheme compared to the baseline(Write Back) almost remains the same. This is because ASIT extra writes are dependent on the application behavior(number of data writes) rather than the locality in the cache. In other two schemes(AGIT read and AGIT plus), the performance depends more on the availability of the metadata inside the cache(due to less counter reads in AGIT read with higher cache size and less eviction in AGIT plus with higher cache size).

7 RELATED WORK

In this section, we discuss the prior works related to Non-Volatile Memory recovery time and crash consistency. The most related work to our paper is selective atomicity [8] and Osiris [7]. In selective atomicity, the authors propose an API for the programmers to selectively persist counters and ensure atomicity through a write queue and Ready-Bit. In order to reduce the overhead of ensuring atomicity of all counters, the paper proposes selective counter atomicity that makes sure of atomicity of only a few counters and does not guarantee atomicity for others. Osiris shows that selective counter atomicity, since not protecting the majority of counters, could result in replay attacks as stale values of counters may occur for these counters after a crash. Osiris discusses the recoverability of all counters using Error Correcting mechanism (ECC bits) of NVDIMM. Each data cacheline, along with its ECC is encrypted using a counter before stored in the NVM. In addition to that, Osiris introduces a stop-loss mechanism for the counter update. Combining these two methods, Osiris can recover the counter across crashes since counters are now recoverable from the data itself. A concurrent work to *Anubis* is *Triad-NVM*[24]. Triad-NVM addresses recovery of general Merkle Trees on systems with both persistent and non-persistent regions. Moreover, Triad-NVM aims at providing trade-offs between resilience, recovery time and performance. The major difference between *Anubis* and *Triad-NVM* is that *Anubis* aims at ultra-low recovery time and additionally enables recovery of SGX-style Merkle Tree, whereas Triad-NVM is limited to general Merkle Tree and providing design trade-offs between resilience, recovery time, and performance.

There are several state-of-the-art works done in NVM security and persistence [18–20, 25–29] without considering the crash-consistency and recovery that discusses to optimize the run-time overhead of implementing security to Non-Volatile-Memory. Most works employ counter-mode encryption for encrypting data and Merkle Tree for ensuring integrity. However, to the best of our knowledge, none of the works discuss the recovery and crash-consistency of integrity protected systems. SecPM [28] proposes a write-through mechanism for counter cache that tries to combine multiple updates of counters to a single write to memory, however, does not ensure recovery for SGX and incurs significant recovery time as in Osiris. To the best of our knowledge, we are the first to discuss the reduction of recovery time of secure non-volatile memory and recovery mechanism that seamlessly works with different integrity protection schemes.

8 CONCLUSION

Anubis bridges the gap between recoverability and high performance in secure Non-Volatile Memories. Our solution can be seamlessly integrated into various secure and integrity protected systems including Intel SGX. *Anubis* can achieve significant improvement in recovery time and incur only 3.4% overhead when implemented in general Merkle tree integrity-protected system, and only 7.9% in complicated SGX-like integrity-protected system. In addition to that, *Anubis* removes the memory size barrier in recovery time and instead makes the recovery time a function of counter cache and Merkle Tree cache size. In summary, with minimal performance overheads, we can achieve recoverability of complicated trees and a recovery time that is less than a second.

REFERENCES

[1] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, pp. 143–143, Jan. 2010.

[2] Z. Li, R. Zhou, and T. Li, "Exploring high-performance and energy proportional interface for phase change memory systems," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, (Washington, DC, USA), pp. 210–221, IEEE Computer Society, 2013.

[3] T.-Y. L. et al., "A 130.7mm2 2-layer 32gb reram memory device in 24nm technology," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2013.

[4] J. Y. P. Zhou, B. Zhao and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *36th annual international symposium on Computer architecture - ISCA'09*, 2009.

[5] E. C. et al., "Advances and future prospects of spin-transfer torque random access memory," in *IEEE Transactions on Magnetics*, Jun. 2010.

[6] M. P. B. R. Chenyu Yan, D. Englender and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," in *33rd International Symposium on Computer Architecture, ISCA'06*, 2006.

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

[8] J. R. S. Liu, A. Kolli and S. Khan, "Crash consistency in encrypted non-volatile main memory systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[9] M. P. B. Rogers, S. Chhabra 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.

[10] "Enhancing High-Performance Computing with Persistent Memory Technology." https://software.intel.com/en-us/articles/enhancing-high-performance-computing-with-persistent-memory-technology. Accessed: 2018-12-07.

[11] "Amazon.com Goes Down, Loses $66,240 Per Minute." https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#68dc6b6b495c. Accessed: 2019-02-19.

[12] A. Awad, P. Manadhata, S. Haber, Y. Solihin, and W. Horne, "Silent shredder: Zero-cost shredding for secure non-volatile main memory controllers," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 263–276, ACM, 2016.

[13] A. Ruddof, "Deprecating the pcommit instruction," 2016.

[14] S. J. Edirisooriya, S. R. Nagesh, B. R. Monson, and P. Kumar, "Method and apparatus for completing pending write requests to volatile memory prior to transitioning to self-refresh mode," Feb. 9 2017. US Patent App. 14/816,445.

[15] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, aug 2011.

[16] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 1–17, sep 2006.

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

[18] M. Taassori, A. Shafiee, and R. Balasubramonian, "Vault: Reducing paging overheads in sgx with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 665–678, 2018.

[19] 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 *Microarchitecture (MICRO), 2018 51st Annual IEEE/ACM International Symposium on*, 2018.

[20] 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)*, pp. 454–465, IEEE, 2018.

[21] M. M. Swift, "Towards o(1) memory," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, (New York, NY, USA), pp. 7–11, ACM, 2017.

[22] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 2–13, ACM, 2009.

[23] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[24] A. Awad, M. Ye, Y. Solihin, L. Njilla, and K. Abu Zubair, "Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[25] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *SIGPLAN Not.*, vol. 46, pp. 105–118, Mar. 2011.

[26] 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 *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 421–432, ACM, 2013.

[27] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 481–493, IEEE, 2017.

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

[29] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 672–685, ACM, 2015.