PS-ORAM: Efficient Crash Consistency Support for Oblivious RAM on NVM

Gang Liu*
liug@hnu.edu.cn
College of Computer Science and Electronic Engineering,
Hunan University
Changsha, Hunan, China

Kenli Li†
lkl@hnu.edu.cn
College of Computer Science and Electronic Engineering,
Hunan University
Changsha, Hunan, China

Zheng Xiao
zxiao@hnu.edu.cn
College of Computer Science and Electronic Engineering,
Hunan University
Changsha, Hunan, China

Rujia Wang*
rwang67@iit.edu
Computer Science Department,
Illinois Institute of Technology
Chicago, Illinois, USA

ABSTRACT

Oblivious RAM (ORAM) is a provable secure primitive to prevent access pattern leakage on the memory bus. By randomly remapping the data blocks and accessing redundant blocks, ORAM prevents access pattern leakage through obfuscation. Byte-addressable non-volatile memory (NVM) is considered as the candidate for main memory due to its better scalability, competitive performance, and persistent data store. While there is much prior work focusing on improving ORAM's performance on the conventional DRAM-based memory system, when the memory technology shifts to use NVM, ensuring an efficient crash-consistent ORAM is needed for security, correctness, and performance. Directly using traditional software-based crash consistency support for ORAM system is not only expensive but also insecure.

In this work, we study how to persist ORAM construction with an NVM-based memory system. To support crash consistency without damaging ORAM system security and compromising the performance, we propose PS-ORAM. PS-ORAM consists of a novel ORAM controller design and a set of ORAM access protocols that support crash consistency. We evaluate PS-ORAM with the system without crash consistency support, non-recursive and recursive PS-ORAM only incurs 4.29% and 3.65% additional performance overhead. The results show that PS-ORAM not only supports effective crash consistency with minimal performance and hardware overhead but also is friendly to NVM lifetime.

*This work was done when Gang Liu was a visiting student at Illinois Insitute of Technology. Both authors contributed equally to this research.
†Corresponding author.

ISCA '22, June 18–22, 2022, New York City, NY

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527425

CCS CONCEPTS

• **Computer systems organization** → **Architectures**.

KEYWORDS

Crash consistency, NVM, ORAM, Persistence, Security

ACM Reference Format:
Gang Liu, Kenli Li, Zheng Xiao, and Rujia Wang. 2022. PS-ORAM: Efficient Crash Consistency Support for Oblivious RAM on NVM. In *Proceedings of The 49th Annual International Symposium on Computer Architecture (ISCA '22).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3470496.3527425

1 INTRODUCTION

Protecting the security and privacy of the data and program running on a shared system is never easy. There is an increasing need for system designers to consider security and privacy protection in addition to performance. There are a lot of efforts from the industry and academia designing secure hardware to give the system a root-of-trust. For example, TPM [8], SGX [34], XOM [38], Trustzone [44] and SME [35], process sensitive data through data encryption and integrity check, or reserve a protected region that cannot be tampered, which effectively prevent adversaries from revealing the plaintext or compromising the data easily. However, the protections are still mainly using encryption and integrity check, which is far from enough. For example, attackers are able to probe sensitive information from victim applications through various side channels, such as the timing information, the power usage and the memory access pattern can be exploited by malicious adversaries to infer sensitive information. Among them, memory access pattern leakage refers to that the adversaries can utilize the temporal and spatial information on the memory address bus to correlate the program's control flow graph [72], the searchable encryption database [31], or even the neural network structure [26, 27].

The cryptographic community proposed Oblivious RAM (ORAM) [22, 23] to address the memory access pattern leakage. The ultimate goal of ORAM is to hide the program access pattern by adding redundant blocks and periodically reshuffling the data in memory. In this way, the attacker will be not able to guess whether the

program is accessing the same or a different data, whether the access is a read or a write, whether we are repeatedly accessing a hot region, etc. The efficiency of ORAM family has improved significantly in recent years. Tree-based ORAM, such as Path ORAM [58], has become one of the mainstream ORAM protocols that people adopt to use on main memory systems[20, 53] with trusted processor. There are extensive research works focus on improving the performance of ORAM on DRAM-based memory systems [11, 13, 50, 63, 64, 71].

We are seeing the scalability issues of DRAM technology and are in the transition to emerging non-volatile memory (NVM) technology. For example, 3dXpoint based Optane memory [24] has already been released to the public; future computing systems such as memory-centric computing architectures [9, 36] use NVM as their unified memory backend. Compared to DRAM, NVM provides natural benefits such as non-volatility, persistency, and high-density. When NVM is architected as persistent memory, it is crucial to maintain crash consistency for data [7, 21, 40, 47, 66]. The specific requirement to address crash consistency is that data (e.g., application data, and configuration, metadata) must be recoverable even if the system power fails or the system crashes [33, 43].

On the other hand, NVM based memory system still faces security challenges like DRAM, such as the information leakage on the memory bus through the access patterns. Applications like collaborative file editing [60] (e.g., Dropbox-like applications) require both security features that protect against access pattern leakage and data crash consistency. Therefore, an NVM-based ORAM system could bring benefits from the two worlds. While some prior works start to address this issue [6, 12, 14, 46], they either work on a different threat model[6], or emphasis on write access overhead [46], or provide a less secure solution [12, 14]. None of the prior works consider the crash consistency problem of ORAM when it is being implemented on NVM. We find that, traditional software-based solutions, such as logging [17, 62] or copy-on-write mechanism (CoW) [18, 61], can only handle general data recovery well; however, such approaches cannot work well with NVM-based secure memory systems for two reasons(details in Section 2.5). First, software-based (e.g., logging or CoW) support for crash consistency mechanisms are inefficient [40, 47]. Second, it may lead to information leakage and break security guarantee. Recently, several NVM-based secure memory systems were proposed with encryption[66] and integrity check [40] support. We are motivated to revisit the crash consistency problem in the presence of ORAM construction and protocol, and further enhance the family of crash-consistent secure NVM systems.

In this work, we study the crash consistency problem when we implement ORAM protocols with the NVM system for the first time. By improving the ORAM hardware architecture and software protocol, we propose an end-to-end PS-ORAM architecture. PS-ORAM system can persistently store ORAM-related data in NVM while solving the crash consistency problem without leaking more information. We first analyze the different components on the ORAM controller to determine the content that needs synchronous persistency and data consistency in Section 2. Then, we analyze persistent atomic access and present different case studies that show what happens if data or other metadata is not persisted during a crash, and analyze the challenges of the problem and the system design goals in Section 3. Next, we present our core design that minimizes the performance overhead due to the persistent write-back and propose an efficient and secure write-back scheme in Section 4. Finally, we evaluate our design in terms of performance, write traffic in Section 5.

2 BACKGROUND AND MOTIVATION

In this section, we first describe the threat model. Second, we introduce the basics of ORAM and NVM. Then, we discuss the problems of traditional software-based persistence methods. Lastly, we describe how ORAM could be implemented on NVM based system.

2.1 Threat Model

We follow the conventional Trusted Computing Base (TCB) boundary and assume that the system equips with a secure and tamper-resistance processor capable of computing without information leakage [48, 50, 58, 71]. Everything on-chip is considered within the TCB boundary. The off-chip main memory system is vulnerable to access pattern attacks, such as physically monitoring the visible signals on the printed circuit boards (including the motherboard and memory modules). The address bus, the command bus, and the data bus are separate from commodity DDR DIMMs in the system. As a result, the memory controller sends out the address and command in cleartext. Therefore, the attacks can be done with physical access to the bus [26, 37] or through side-channel analysis [26, 27]. By observing the access patterns such as access frequency, access type (read or write), and also the repeatability of accessing the same location, the attacker can obtain some leaked sensitive information in the program [31].

In some system settings, part of the main memory system can be considered as protected and free from most of security attacks. For example, with SGX [34], a small region in the memory called EPC can store pages safely. With cmov-based operation, the accesses to EPC region can be considered as oblivious too[1, 53]. In this work, we discuss implementations under the two assumptions: 1) memory is fully untrusted; 2) memory has a partially trusted region. The different assumptions will change how ORAM metadata can be persisted without leaking information. We discuss this issue in detail in Section 4.4.

2.2 ORAM Basics

ORAM [22] is a security primitive that can hide the program's access pattern and accordingly eliminate information leakage. ORAM's basic idea is to access more blocks than the actual data we need, and shuffle the address space so that the access address becomes random. With the ORAM controller in the secure processor, one memory access from the program is translated into an ORAM-protected sequence. ORAM protocol guarantees that any two ORAM access sequences are computationally indistinguishable. In other words, ORAM physical access pattern and the original logical access pattern are independent, which hides the actual data address with the ORAM obfuscation. Since all ORAM access sequences are indistinguishable, an attacker cannot extract sensitive information through the access pattern. Tree-based ORAM schemes, such as Path ORAM [58] and Ring ORAM [48], have improved the overall access and reshuffle efficiency greatly through cryptographic innovations. In

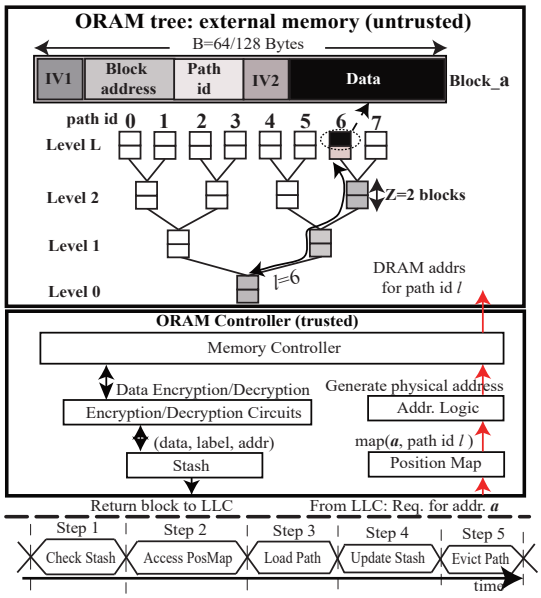


Figure 1: Path ORAM construction and access protocol

this work, we focus on one of the most representative tree-based ORAMs, Path ORAM [58], which is the building block of many data oblivious frameworks, such as Obliviate [2], Taostore [52] and Zerotrace [53].

2.2.1 Path ORAM Construction. Logically, Path ORAM reorganizes the external memory into a binary tree (we refer to as the ORAM tree). Upon a memory request from the LLC, a full path of data blocks is fetched, as shown in Figure 1. The node in the ORAM tree is called a bucket and can hold Z data blocks. The height of the ORAM tree is noted as L. In Figure 1, we show an ORAM tree with 4 levels ($L = 3$), and the bucket size equals to 2 ($Z = 2$). Each block inside the bucket contains the encrypted data content and a header that tracks the program address, path id, and initialization vectors (IV) used with AES counter mode encryption. Dummy blocks are marked with a special program address $\perp$. Following [20], IV1 is used to encrypt the block's header, while IV2 is used to encrypt the data content.

On the trusted side, the ORAM controller converts the regular memory access pattern into ORAM sequences. ORAM controller mainly includes a position map (PosMap), a stash, address translation logic, and encryption/decryption circuit. The PosMap is a lookup table that stores the path id (leaf label) for a given logical address. The stash is a small buffer that can hold a small number of data blocks [50] during the path accesses. The obliviousness of the access pattern is achieved by randomly remapping the path id of a data block after each access.

2.2.2 Path ORAM Access Protocol. Next, we discuss the Path ORAM access protocol. Given a memory request $a = (addr, read/write, data)$ for data block a, the access steps of $ORAM(a)$ are as below:

① **Check Stash:** Check if the block a is in the stash. If hit, fetch the data block to the processor if it is a read, or update the value if it is a write. If it is a miss, proceed to the next step.

② **Access PosMap:** The actual physical memory location of block a is determined by checking the PosMap with $addr$, and a path id l is returned. Then, randomly generate and update a new path id l' for the accessed block a.

③ **Load Path:** Load all blocks on path l from the ORAM tree in the memory to the stash, decrypt them and find the block a. Then, return the block a to the processor if it's a read operation, or update the value in the stash if it's a write operation.

在stash中

④ **Update Stash:** The path id of the block a in the stash also needs to be updated to l'. In this case, data blocks in the stash have the most up-to-date value and path id.

⑤ **Evict Path:** Evict data in the stash back to memory on path l. The basic rule of eviction is to fill as many blocks as possible that can be written to path l. If the real blocks are not enough, then pad with dummy blocks.

2.3 Persistent System with NVM

Emerging NVM technologies, such as Phase-Change Memory (PCM), Spin-Transfer Torque (STT-RAM), and Memristor, are considered candidates for replacing conventional technologies such as DRAM and NAND Flash. The Micron and Intel 3dXpoint-based Optane [29] has shown competitive performance, density and scalability with conventional technology. When used as main memory, NVMs may provide *persistent memory*, where regular store instructions can be used to make persistent changes to data structures to keep them safe from crashes or failures. A great number of research efforts have sought to optimize recoverable or crash-consistent software (e.g., databases [4, 5], file systems [15, 55], key-value stores [65, 67]) for NVMs.

On the other hand, NVM systems still suffer from various security vulnerabilities. To provide data confidentiality, NVM can utilize lightweight encryption schemes [59, 69]; to detect and fix integrity issues, adopting Merkle tree and support its persistent updates have been recently studied [7, 66, 73]. Access pattern leakage is another degree of vulnerability, and we can add obfuscation with the help of ORAM[46].

2.4 Crash-consistent ORAM Systems

While the main memory could be replaced with NVM, the on-chip cache and buffers still use volatile memory for better performance. To ensure the on-chip content can be flushed back to the NVM, Intel Asynchronous DRAM Refresh (ADR) [32] provides write pending queues (WPQs) as on-chip persistence domain. In the event of crash, the content in the WPQs can be persisted to NVM for crash consistency. However, when there is an ORAM controller sit between the WPQ and the LLC, we need to consider how to persist the content in stash and PosMap, as they are not part of the persistence domain yet.

After several ORAM accesses, a small number of data blocks will remain in the volatile stash. Such data blocks could contain the most up-to-date values for a given logical address. Consider that a failure happens during the execution, such content in the stash may be lost before they are written back to the NVM-based ORAM tree. The loss of data in the stash not only causes a crash consistency problem but also causes the system to fail to correctly recover lost data blocks. Similarly, the PosMap contains mapping information that determines where to locate a block in the main memory. Each

data block is given a path id, and it is not only associated with the block (in the header), but also stores in the PosMap. As discussed in section 2.2.2, the updates on path id happen on multiple steps. If the PosMap is volatile, we will not be able to locate the block of interest in the main memory.

Furthermore, we identify that if the ORAM access needs to be recoverable, the data buffered in the stash and the PosMap needs to be persisted atomically. Otherwise, data inconsistencies could happen when we try to recover from a crash. We discuss the details of the writeback inconsistencies and design requirements in the next section.

2.5 Limitations with Software-based Crash Consistency Support

Although traditional software-based mechanisms can be used to support crash consistency in general, it is challenging to apply it to ORAM systems for several reasons. For example, the logging-based system [17, 62] maintains a backup copy of the original data in the log, and the log system redoes log (store new data) or undoes log (store old data). Logging consumes much more NVM capacity than the original data, because each log entry is an original tuple of data and corresponding metadata (e.g., counter value, data address, etc.), and typically each memory record must be logged [17, 62]. Therefore, directly adopting logging-based schemes to support the crash consistency of the ORAM system is impractical: it will cause significant performance loss, slow recovery, and more memory space overhead. Similarly, a copy-on-write-based (CoW) system [18, 61] always creates a new copy of the data to be updated. The disadvantage of CoW is that the copy operation cost is expensive and cause long stall time [56]. Since ORAM reads and writes multiple blocks along the path, if every accessed data block is to be copied, it will not only cause memory capacity overhead but also lead to more serious performance loss. Also, additional NVM bandwidth is required due to the copy of redundant unmodified data blocks [57]. There are abundant dummy blocks accesses in ORAM system, and backing up these dummy blocks are useless and causing lifetime reduction of NVM.

Additionally, software-based approaches may cause information leakage, which undermines the security protection of ORAM. For example, if the log is stored without protection, then the attacker will obtain the related access pattern or data information by peeking at the log, which will cause information leakage.

2.6 Design Challenges and Scope of This Work

To summarize, it is challenging to implement ORAM on NVM for three reasons: 1) ORAM is expensive in terms of memory access overhead; 2) simply replacing the memory device to NVM cannot provide the ORAM accesses with crash consistency; 3) Using software-based approach to support ORAM crash consistency could lead to huge performance loss and security problems.

In this work, we focus on enabling persistent ORAM system with low overhead, without leaking additional information. We believe that to achieve provable secure access pattern obfuscation, ORAM is required, and the cost of ORAM protocol can be further optimized with the cryptographic innovation. On the other hand, ensuring crash consistency for the ORAM system is a critical problem to be solved by the computer architecture community when the memory system shifts to NVM technology.

3 DESIGN REQUIREMENTS FOR CRASH RECOVERABLE ORAM

In this section, we discuss the design requirements for a recoverable persistent ORAM system. Simply replacing the main memory technology to NVM cannot guarantee consistent recovery. An ideal case would be that all on-chip buffers are built from NVM to write to the stash or position map is persistent immediately. However, as most of the on-chip components are still considered volatile, we identify a need to properly handle the volatile data in the ORAM controller to make the overall ORAM system persistent.

3.1 Consistent Metadata Update

The ORAM accesses not only require updating the data block, but also the metadata associated with it, including the header and the position map entry. Here, we define the *consistent metadata update* requirement as follows: when there is a crash happening at any ORAM access step, we can restart the ORAM access by identifying the target data block location in the NVM again. In other words, the path id information and other metadata should not be lost.

Figure 2 demonstrates why consistent metadata update is desired. In step 2 of an ORAM access, a new path id is randomly generated for the target block, and the corresponding entry in the PosMap is updated. If the metadata is not persisted consistently, any crash happens after step 2 would possibly cause data inconsistency since the path id is changed. We discuss the details by several case studies in Section 3.3.

3.2 Atomic ORAM Accesses to NVM

Except for the consistent metadata updates, another design requirement for persistent ORAM is to preserve the access atomicity. Here, we define the *ORAM access atomicity* as follows: The data in the stash and the metadata in the PosMap should reach persistency in an atomic way. If one of them is persisted while the other is not, the continued ORAM access is then out-of-sync.

The reason to have atomic ORAM access is that the metadata and the data correspond to the same actual memory request. On a system failure, if only the content in the stash is persisted by writing back to the NVM-ORAM tree, the data content in the NVM-ORAM tree would be overwritten. In this case, if the PosMap entries are not persisted yet, it is impossible to locate the new path id where the data is located. A reverse example is if the metadata in PosMap is persisted, but the stash data is not, based on the new path id in PosMap, it is impossible to recover the lost data in the stash. We also discuss the details of why atomicity is needed in Section 3.3.

3.3 Case Studies on Crash Recoverability

To summarize, to ensure a recoverable ORAM access after a crash, we need to ensure the following requirements are met:

a) Ensure that the accessed data blocks in the NVM-ORAM tree are not lost during a crash. Data blocks in the stash that have not been evicted back into the NVM-ORAM tree can not be lost.

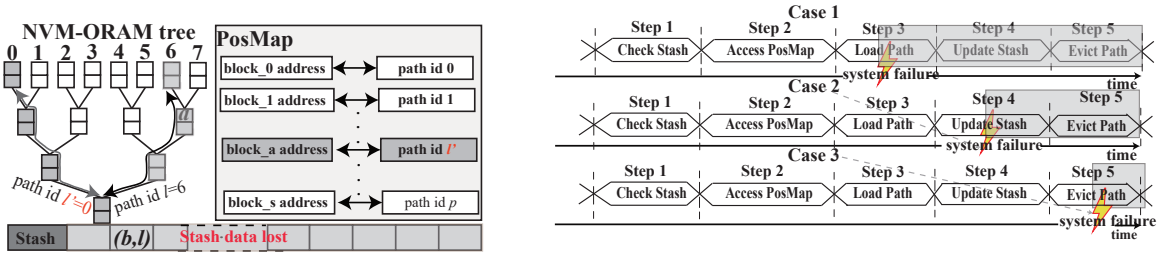


Figure 2: Step-by-Step diagram of NVM-based ORAM systems crash

b) The address and path id contained in each block evicted from stash to NVM-ORAM tree should be consistent with the metadata stored in the updated (persistent) PosMap, that is, consistent updates.

c) The updated path ids of the accessed data in the PosMap, the data in the stash, should all reach the NVM atomically. Otherwise, there is a mismatch between the data persistency and metadata persistency.

Figure 2 shows an example that when the requirements are not met, during a crash, the NVM-based ORAM system could result in inconsistent status. We assume that n ORAM accesses have been performed, so some data blocks remain in the stash, e.g., block b. At the time of the crash, we are performing the $(n+1)$-th ORAM access. At step 2, the PosMap update is completed, i.e., the block a is mapped to a new path id l' in PosMap. Then, on step 3-5, we could observe different types of inconsistencies due to the path id remapping process.

Case 1: If the crash occurs in step 3 during the ORAM access, since the path id of block a in PosMap has been updated ($l \rightarrow l'$), and block b has not been written back from the stash, no matter this metadata update is persisted or not, it violates the consistency and atomicity requirements.

a) If the PosMap data has not been persisted, then the metadata in PosMap is restored to the last persistent state. In the worst case, if the metadata in PosMap is not persisted after performing n ORAM accesses, it returns to the initial state when the NVM-based ORAM system starts. The data blocks distribution in the NVM-ORAM tree has already changed after n ORAM accesses. If the program continues to perform ORAM access based on the old metadata in the PosMap, it will cause access errors.

b) If the metadata in PosMap has been persisted after the update, then we can always retrieve the most up-to-date metadata when a crash happens. In this case, we retrieve path l' for block a. However, a is never written back to path l'. Therefore, even with the persisted new path id, we cannot fetch the data from NVM again.

c) Regardless of whether the metadata in PosMap has been persisted, the data blocks stored in the volatile stash are lost, including the dirty ones with new values.

Case 2: If the system crash occurs in step 4, the good news is that block a is already fetched and path l has been fetched into the stash so that this particular access may succeed. However, similar to the case 1, the content in the stash would be all lost.

Case 3: If the system crash occurs in step 5 of ORAM access, or before the next ORAM access, it may cause inconsistent data updates. Step 5 is to write data back to the NVM-ORAM tree, and the operation is a natural data persistency operation. We discuss the following scenarios that may happen:

a) The stash content is lost, similar to case 1 and 2.

b) During the path eviction process, it is possible that some data blocks have been written back to the NVM-ORAM tree while some are not. This will cause non-atomic data write-backs to the ORAM tree and overwrite some of the real data blocks.

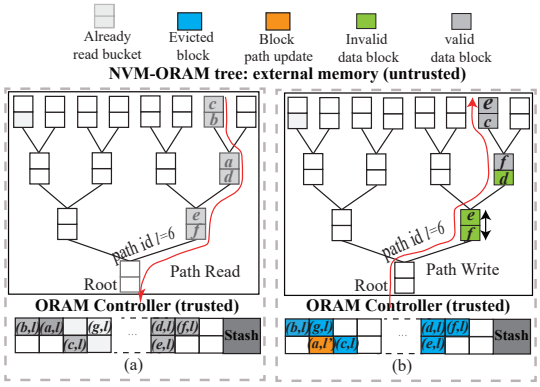


Figure 3: Data overwritten by partially path writeback

Figure 3(a) shows the data fetched from the NVM-ORAM tree into the stash. In the original write back process, the target block a in the NVM-ORAM tree is overwritten by the write back process, and the position of other blocks also changes, as shown in Figure 3(b). Obviously, if the system crashes after the completion of the writeback and before the next ORAM access, block a will be lost and unrecoverable. If a system crash occurs during write back, more data blocks may be lost. If the system crashes when writing back data block g, the data in the stash is lost, and the data blocks a, and b on the NVM-ORAM tree have been overwritten by blocks c and f, respectively. Data blocks a and b are lost and cannot be recovered, as shown in Figure 3(a).

Through the in-depth case study, we understand the design requirements for a recoverable persistent ORAM system. We do not want the stash content to be lost; meanwhile, we would like the updates on PosMap to be consistent with the contents in the stash; further, we would like to ensure the atomicity of data and metadata writebacks.

4 THE DESIGN OF CRASH CONSISTENCY ORAM

In this section, we present PS-ORAM, a novel crash-consistent architecture designed for persistent ORAM. The PS-ORAM includes

a new ORAM controller architecture that requires the necessary hardware support to protect the accessed data blocks from loss and consistency with metadata updates in PosMap. Further, we incorporate the persistent atomic writebacks into the ORAM protocol and analyze how the crash consistency can be achieved. The hardware and protocol innovations ensure that the persistency is done correctly and do not destroy the ORAM obfuscation capability.

4.1 PS-ORAM Architecture Overview

To protect the crash consistency of the NVM-based ORAM systems, we propose PS-ORAM architecture with the design requirements discussed above. The overview of PS-ORAM architecture is shown in Figure 4. Besides the basic components of the existing ORAM controller (i.e, a stash and a PosMap), the following components are needed to ensure crash consistency: a *Temporary PosMap*, and the persistence domain which includes a *Drainer* and *write pending queues (WPQs)*. Note that the persistent domain (supported by ADR or eADR) is standard in many crash-consistent architectures [21, 40, 47, 51, 66]. During a crash, the contents in the persistence domain can still be flushed back to the NVM atomically.

Here, the drainer is connected to the encryption/decrypting circuit and dispenses the data evicted from the stash and temporary PosMap to the two corresponding WPQs. Also, the drainer is responsible for issue control the "start" and "end" signals to control the WPQs receiving data and the signals persisted to the NVM.

The temporary PosMap stores the reassigned path ids of the accessed target data blocks. Specifically, according to the access protocol of Path ORAM, we know that each time the ORAM controller touches a target data block, a new path id is assigned to it (see Step 2 in Section 2.2.2). At this time, the address of the accessed target blocks and the corresponding new path id will be stored in the temporary PosMap to wait for the data to be persisted. As long as the temporary PosMap is not merged with the main PosMap, we do not overwrite the original path id that has been persisted already. The data blocks WPQ is used for storing evicted data blocks from stash. Each time when the data blocks are evicted from the stash, they then enter the data blocks WPQ to achieve atomic persistence. The ***PosMap WPQ*** is used to persist recently changed path ids corresponding to the data blocks evicted from the stash. The content in ***PosMap WPQ*** comes from the temporary PosMap.

4.2 PS-ORAM Workflow

We then describe the PS-ORAM workflow in this section. To provide the ORAM system with crash consistency, we revisit the basic Path ORAM workflow and carefully integrate the persistent operations during the ORAM access. The updated ORAM access protocol still follows the main workflow without leaking information. The circled numbers in Figure 4 represent the dataflow corresponding to each step of the PS-ORAM protocol.

4.2.1 PS-ORAM access protocol. Given a memory request $a = (addr, read/write, data)$ for data block a, the five access steps of *PS-ORAM(a)* are as below:

① **Check Stash:** This step remains unchanged as Step 1 in Section 2.2. If the block a is not in the stash, proceed to the next step.

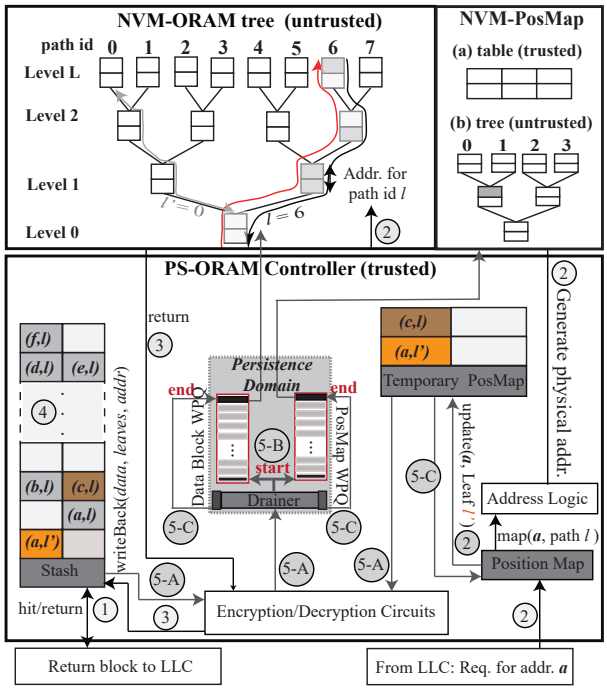


Figure 4: PS-ORAM system architecture.

② **Access PosMap and Backup Label:** Similarly, we check PosMap with $addr$, and l is returned as the target path id. Then, the data block a is remapped to the new path id label l'. Instead of overwriting the (a, l') directly in the PosMap, PS-ORAM stores the new path id l' in the *temporary PosMap*.

③ **Load Path:** This step is unchanged from the original Step 3 in Section 2.2.

④ **Update Stash and Backup Data:** Since the new label l' has been reassigned to the target data block a in step 2, the path id in the header of the data block a fetched from the NVM-ORAM tree to stash is now updated to l'. Meanwhile, the original data block (a, l) is copied in the stash as a backup block (similar to the concept of shadow block in [70]). In this case, we can make sure the backup block will be written back to path l during the eviction. Later on, when block a is evicted to path l', the backup data block will be invalid automatically[1].

⑤ **PS-ORAM Eviction:** Lastly, the eviction needs to be done properly to ensure crash consistency. We describe the details of the persistent evict path operation in Section 4.2.2.

Note that the main function of the backup data block generated in Step 4 of PS-ORAM access is to recover the data block lost after the crashed system. We analyze how to recover lost data in Section 4.3.

4.2.2 PS-ORAM eviction in detail. The PS-ORAM eviction is the main step of writing the data or metadata from volatile on-chip components back to the persistent NVM system. We show the substeps of eviction as below.

[1]When read path l again into the stash, the path id of backup block will mismatch the up-to-date one; therefore, the block can be regarded as a dummy block.

- **Step 5-A (Encrypt evicted blocks)** The data blocks that need to be written back from the stash are identified first. Because PS-ORAM loads the path l in Step 3, the eviction path is also l. In Figure 4, the gray and brown blocks are identified and they will be encrypted. Note that the backup block (a, l) is also included as an eviction candidate. Meanwhile, if the data block's path id has been changed, the corresponding dirty metadata entries in the temporary PosMap are identified[2]. In this example, the entry (c, l) is identified and will be encrypted. The block c was previously fetched and path l is its new path id.
- **Step 5-B (Push data into WPQs)** Once the eviction data blocks and metadata are ready from encryption, the drainer sends the "start" signal, and the candidate data blocks and PosMap entries are loaded into the two corresponding WPQs. Note that the "start" signal controls both WPQs, as such, the data and metadata can be load into the persistence domain atomically.
- **Step 5-C (Write to NVM)** When the data and metadata for this eviction round are all in the WPQ, an "end" signal is sent to both WPQs, meaning that the ORAM eviction is now atomic. Then the two WPQs are flushed back to the NVM-ORAM tree and the PosMap in the NVM. Note that the storage format of PosMap depend on the threat model: if the PosMap is kept in a trusted region in the NVM, then the write back can be done through direct updates to the table; if the PosMap is not kept in a trusted NVM region, recursive PosMap is needed to keep the writebacks secure. Figure 4 shows the two formats of storing PosMap in memory securely. We discuss the options to implement the two ***PosMap WPQ*** flushing cases in Section 4.4.

Tracking the dirty PosMap entries and only putting them into the WPQ can greatly reduce the performance overhead, by removing most of the redundant metadata writes. Meanwhile, PS-ORAM can still achieve consistent metadata update and atomic ORAM accesses to NVM. Otherwise, for all $Z \cdot (L+1)$ blocks on the path, we need to flush $Z \cdot (L+1)$ PosMap entries as well (refers to Na?ve-PS-ORAM in our experiments).

4.2.3 Discussions on persistence domain implementation choices. We now discuss how PS-ORAM design can adapt to different persistence domain technologies.

ADR-supported WPQs. We first describe the persistence domain (WPQs) supported by ADR technology. Ideally, the sizes of the ***data block WPQ*** and ***PosMap WPQ*** should be large enough to hold the real data blocks and metadata of one full path access, which depend on the ORAM tree size. Considering the worst case that the all data blocks on the evicted path are real blocks, the ***data block WPQ*** needs to store $Z \cdot (L+1)$ data blocks, and the ***PosMap WPQ*** needs to store $Z \cdot (L+1)$ path ids. Considering the ORAM parameters in Section 5.1, the size of ***data block WPQ*** is 96-entry (6144B), and the size of ***PosMap WPQ*** is 96-entry (672B). The WPQ sizes in the persistence domain is about 2x of the current size with ADR technology[21, 28, 39, 73], and 33.13% more than SCA[40]. Note that, the dummy blocks in the ORAM tree account for half of the total capacity[50, 58]. Therefore, the number of real blocks on each path fluctuates around $Z \cdot (L+1)/2$ entries, and the WPQ sizes can be reduced to half as well.

Limited persistence domain with few WPQ entries. If the WPQ sizes are too small to hold $Z \cdot (L+1)$ entries, we need to slightly modify the PS-ORAM eviction process to provide guaranteed crash consistency by tracking the write orders of real blocks. To prevent the data in the NVM from being overwritten by the write-back blocks (cases in the Figure 3), we need to enforce the order of writing blocks back. For example, in Figure 3, the evicted block e overwrites c, and c overwrites b. If we only have a small WPQ, then evicted real blocks should follow such an order: $\{e \rightarrow c \rightarrow b \rightarrow \cdots\}$. Additional dummy blocks can be inserted in between of real blocks during the eviction from the WPQ to the NVM.

Extended persistence domain with eADR. The eADR technology can extend the persistent domain capacity to the cache hierarchy [30, 54] and reduce the management overhead for general data persistency. PS-ORAM protocols can work seamlessly with the eADR technology to support both crash consistency and security. The WPQs in the ideal case can easily fit into the eADR supported persistence domain; alternatively, if the WPQs are still limited, we can temporally store a portion of blocks on the eviction path in the eADR domain without losing them.

Note that, simply extending eADR to the entire on-chip buffers of an ORAM system can lead to security issues during a crash. For example, the content in the stash could be flushed directly back to the NVM without following the ORAM protocol, which leads to information leakage. PS-ORAM is still needed in the presence of eADR. eADR has to support the entire ORAM controller and provide extra energy to flush data blocks to gain similar data persistency; however, the overhead is much higher than PS-ORAM. We then show the eADR-based ORAM system (eADR-ORAM) overhead in section 4.2.4.

The impact of WPQ sizes on PS-ORAM performance. The sizes of WPQs do not affect the performance of proposed PS-ORAM system. The reason is that the WPQs are not traversed when ORAM is accessed; only the stash and PosMaps are used to look up a block. Therefore, no read or write merge may happen in the WPQs. Also, we do not relax the data persistence model – all modified data blocks are persisted to NVM in-order without coalescing.

4.2.4 Draining Cost Comparison of PS-ORAM and eADR-ORAM. eADR draining cost depends on the on-chip cache and buffer sizes [3]. In this work, the experimental system configuration is shown in the Table 3. Both the stash and the (temporary) PosMap in ORAM system are volatile using SRAM, so the total on-chip cache and buffer size of the system is 1.0625 + 0.012207 + 192 = 193.07MB. The energy needed to access data in such SRAM cells is estimated to be about 1pJ/Byte[3]. Table 1 shows the estimated energy needed for draining data from different cache levels to NVM. The numbers are derived from the analysis and discussion in [3, 42].

Estimated energy comparison. We assume that in the case of a system crash, eADR-ORAM design needs to provide enough energy to persist the data in the cache, stash and (temporary) PosMap to NVM following ORAM protocol. Table 2 presents the average energy needed to drain data from caches (for eADR-ORAM) and

[2]Writing back all metadata entries of blocks along the path can also achieve the same design goals, with more write-back overhead. We refer it to the Na?ve-PS-ORAM in our experiments (Section 5.1).

Table 1: Energy cost estimation in case of system crashes following [3]

Operation	Energy Cost
Accessing Data from SRAM	1pJ/Byte
Moving data from L1D to NVM	11.839nJ/Byte
Moving data from L2, stash, PosMap and WPQs to NVM	11.228nJ/Byte

from PS-ORAM, based on the cost model discussed in [3]. This energy consumption does not calculate the energy consumption of data block encryption, thus, this assumption produces an optimistic energy data for eADR-ORAM. For different WPQs size settings (96 and 4-entries) in PS-ORAM, compare the energy consumed by eADR-ORAM and PS-ORAM when the system crashes to 2.286J and 76.530μJ (2.83μJ), respectively. Despite more realistic estimates, PS-ORAM is 29870x and 807797x more efficient than eADR-ORAM at different WPQ size settings, respectively. The energy cost of PS-ORAM is 5 to 6 orders of magnitude lower than eADR-ORAM.

Estimated draining time. We calculate the data draining time based on [3, 32]. Table 2 shows the average time required for drain data for both eADR-ORAM and PS-ORAM technologies. eADR-ORAM technology takes 4.817ms. In contrast, PS-ORAM only takes only 161.134ns and 6.713ns for different WPQ size settings.

If eADR only supports the energy consumption of flushing the cache and the stash, but does not support ORAM protocol persistence (eADR-cache), the required energy consumption and time are 12.653mJ and 26.638μs respectively, which are about 165x higher than PS-ORAM, as shown in Table 2.

4.3 Data Recovery Consistency Analysis

In this section, we show how PS-ORAM can guarantee a consistent crash recovery through case studies. We revisit the three cases in Section 3.3 and analyze why the prior issues are addressed.

Case 1: In the original Path ORAM, PosMap has been updated before step 3 of each ORAM access. As a result, when the system crashes during step 3, data blocks stored in the volatile stash are all lost. Therefore, it will cause a crash consistency problem because the data in the volatile stash is not persisted in time.

With PS-ORAM architecture, since step 2 is enhanced, the new path ids of the accessed data blocks are not committed directly into the PosMap but into the temporary PosMap (volatile). Therefore, if the PS-ORAM system crashes in step 3, the data in the temporary PosMap, and stash will all be lost at the same time. During the recovery process, the ORAM controller can re-read this path id before remapping again with consistent path id in the PosMap. Therefore, when performing this ORAM access again, the matching PosMap can still correctly access the data of interest in the original path from the NVM-ORAM tree.

Case 2: When the system crash occurs at step 4 of the ORAM access, the scenario is similar to case 1. The difference is that the ORAM controller has fetched data blocks from a path to stash, so the data blocks on that path are marked as invalid. Invalidate data blocks in the NVM-ORAM tree only happen with some updates on metadata, not the actual data content, therefore, there is no data loss or mismatch happening. During the recovery, the ORAM controller only needs to restore the data that has been marked as invalid to a valid during the read path. Then, the lost data can be recovered from the data content region.

Case 3: If the ORAM system crash occurs in step 5 of the ORAM access or before the next ORAM access, as discussed before, it may cause inconsistency with partial writebacks (either data or metadata). As a result, some valid data along the path are no longer recoverable. Also, lost data in stash and PosMap scenario is similar as Case 1 and 2.

We create the backup block for accessed block and write it back to the original path l together with other data blocks to solve the overwritten problem. At the same time of writing back, PosMap does not update the path id of the target block that has not been evicted from stash, so the target block's original path id is still stored in PosMap (Section 4.2.1 Step 3). If the system crashes at this time, the target blocks that have not been evicted in the stash are lost, but their backup blocks can still be found and restored in the NVM-ORAM tree.

In addition, the added on-chip WPQs can ensure the volatile data in stash and PosMap enter the persistence domain at the same time. We do not need to worry about the content in the stash,and PosMap is gone with a crash.

If the system crashes before the "end" signal is received by the write pending queue, the original data blocks on the write-back path still exist and will not be overwritten, so the data can be recovered. Therefore, with PS-ORAM writeback operation, the data blocks in stash and PosMap can be consistent, and the data blocks lost after the system crash can be effectively recovered.

4.4 Implement and Persist Non-recursive and Recursive PosMap in NVM

PosMap is the key component in ORAM system, as it stores all mapping information for each memory request. Phantom [41] is the first hardware ORAM prototype built on FPGA. Since the FPGA memory is relatively small, the Phantom design stores the entire PosMap on the chip. However, if the ORAM tree size is large, it is hard to store the entire PosMap on-chip. For example, a 4GB ORAM tree with 128bytes and $Z = 4$ requires a 93MB PosMap size [50]. To solve the problem of large PosMap size, recursive ORAM is proposed [19, 49]. In this way, the PosMap in untrusted main memory is also stored as a small ORAM tree, while the on-chip PosMap is a cache for most recently used PosMap entries. Update the PosMap in the memory requires a small ORAM tree write path operation.

A more ideal case would be, the PosMap can be stored in a trusted memory region and any read or write operations to the PosMap are free from most of security vulnerabilities [1, 2, 53]. In this case, a cmov-based oblivious update is desired to further obfuscate the access pattern to the PosMap. The oblivious PosMap update generates fake addresses for all entries in the PosMap, but only the updated entries will be actually written.

In this work, we consider *both* cases of implementing and accessing PosMap on NVM main memory. We implement the recursive ORAM and PosMap accesses following [19] for untrusted memory. Also, we consider the non-recursive PosMap is kept at a on-chip secure region (similar to [41]) and cmov-based PosMap updates [1, 2, 53] can ensure the writebacks are still oblivious.

Table 2: Estimated draining energy and time cost for PS-ORAM vs. eADR.

Technology	eADR		PS-ORAM (WPQ sizes)		Normalized to PS-ORAM (WPQ size=96 / 4)		
System	eADR-cache	eADR-ORAM	96-*entries*	4-*entries*	eADR-cache	eADR-ORAM	PS-ORAM
Energy	12.653mJ	2.286J	76.530μJ	2.83μJ	165× / 4471×	29870× / 807797×	1
Time	26.638μs	4.817ms	161.134ns	6.713ns	165× / 3968×	29894× / 717563×	1

4.5 Apply PS-ORAM to Hybrid Memory System.

The PS-ORAM workflow, such as persistent eviction and in-place data backup, can be applied to the hybrid memory system, when the hybrid memory architecture is clearly defined. When the memory is organized with multiple technologies, how to place the data across NVM and DRAM, how often to persist data from DRAM to NVM, will change the detailed steps of the design. We reserve this direction as our future work.

4.6 Security Analysis

ORAM is designed to hide the original program's memory access pattern, and its security depends on the independence of the label sequence, randomness, and the same length of the access sequence [58]. In PS-ORAM, we modify the step 2,4 and 5 of ORAM access for the add-on persistency. However, we do not modify the random remapping process and the redundant sequences of ORAM access. The added components and data block backup steps all happen on the trusted ORAM controller side. Therefore, the modifications do not leak any access pattern information, or cause stash/ORAM tree capacity overflow.

Claim 1*: Step 2 does not leak additional information.* The backup label operation happens inside of the ORAM controller, which is inside of the trusted boundary.

Claim 2*: Step 4 does not leak additional information or cause overflow.* The backup data block is written back to the original path each time. Therefore, the stash occupancy does not change after each ORAM access. When the block is written back to its new path, the previous copied block is marked as invalid, so occupied memory space is freed again. As a result, we do not increase the stash and ORAM tree overflow probability. A similar use case has been discussed in [70].

Claim 3*: Step 5 does not leak information during the writebacks.* The data blocks written back from WPQ remain the same as the baseline Path ORAM. As for the security of PosMap, in this work, we consider two situations to protect PosMap. When the PosMap is stored in an SGX-like trusted memory region [34], the CMOV-based PosMap update approach [1, 53] is adopted to ensure the obliviousness. On the memory address bus, all entries in the PosMap is touched, but only the ones that require changes are written with new values. If no trusted memory region is available, we store the PosMap recursively [19], and the writing back one path id updates involves a small PosMap ORAM path write. Hence, PS-ORAM PosMap writeback does not introduce additional access pattern leakage.

Claim 4*: The backup block does not leak information when the system crashes.* After the crash, the system will always try to access the last path that contains the backup block again and follows the ORAM protocol. The content of backup block will only be known in the stash after all blocks are read along the path.

Claim 5*: Reordering due to limited WPQ sizes does not leak information.* With a small WPQ size, we need to enforce the order of eviction blocks to ensure no overwritten happens. Such reordering scheme does not leak information, because dummy blocks are inserted to form a full eviction path, and the observed write back addresses are the same as original.

To summarize, PS-ORAM architecture and its access protocol support crash consistency without leaking additional information on access patterns.

5 EVALUATION

In this section, we first describe the relevant settings for experimental evaluation. Then, the designs of the experimental evaluation are described. Finally, the detailed evaluation results of each experimental design are given.

5.1 Methodology

To evaluate our design, we use the cycle-accurate gem5 simulator [10] for on-chip components and NVMain 2.0 [45] for the NVM-based main memory. Table 3 summarizes the configurations of processor, ORAM controller, and main memory. We modeled an in-order core at 3.2GHz[63, 64]. Since we focus on the memory system, using in-order or out-of-order core does not affect the overall memory access overhead. To minimize the possibility of stash overflow, the ORAM utilization rate is set to 50%, following previous works [50, 63, 64, 70, 71]. Therefore, to store 2GB of data, 4GB of NVM is required. Without loss of generality, we use phase-change memory (PCM) [16, 45]. The data block size is set to 64B to match cacheline size [63, 64, 70, 71]. Considering the worst case, the size of the Temporary PosMap (C_{tPos}) is set to 96-entry. We set the ***data block*** and ***PosMap WPQ*** sizes to 96-entry and 96-entry, respectively (hardware overhead details in Section 4.2.3). For other system-related parameters, we use the default values of gem5 and NVMain 2.0. We use 14 workloads from SPEC 2006 [25] benchmark suite in the experiments, following the experimental settings of [21]. The MPKIs of each workload are shown in the Table 4. We use simpoint to collect 5,000,000 samples per trace in each workload. we assume the overall AES encryption latency to be 32 cycles [19, 70], and we overlap fetching data with encryption pad generation [68].

We implement and evaluate four different persistent ORAM system protocols and compare them with the baseline non-recursive/recursive ORAM protocols without data persistency, as described below.

- **Baseline:** It refers to the baseline Path ORAM protocol implementation with NVM system, without data crash consistency. Compare with a non-ORAM system with NVM main

Table 3: Experimental Setting Configurations

(a) **On-chip processor and cache**

Core type/frequency	in-order (1 core), 3.2 GHz
L1 I/D cache L1 read/write	32KB/32KB, 2-way LRU 2/2-cycle
L2 cache L2 read/write	1MB shared, 8-way LRU 20/20-cycle

(b) **ORAM controller**

Data block size	64B
Data ORAM capacity	4GB ($L = 23$)
Block slots per bucket (Z)	4
Stash size (C) [50]	200-entry
Temporary PosMap size (C_{tPos})	96-entry
AES-128 latency	32 cycles [19, 70]

(c) **Persistence domain**

PCM [16, 45]	4GB, 400MHz , $t_{RCD}/t_{WP}/t_{CWD}/t_{WTR}/t_{RP}/t_{CCD}$ =48/60/4/3/1/2
STTRAM[45]	4GB, 400MHz , $t_{RCD}/t_{WP}/t_{CWD}/t_{WTR}/t_{RP}/t_{CCD}$ =14/14/10/5/1/2
WPQs	96/4-entry for ***PosMap WPQ***, 96/4-entry for Data WPQ

Table 4: Workloads and their MPKIs

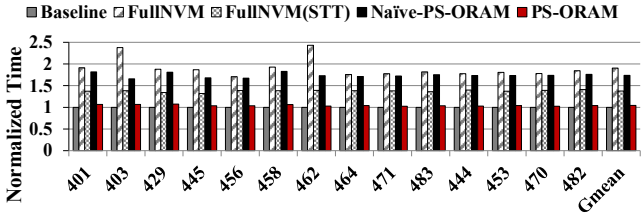
Workload	MPKI	Workload	MPKI
401.bzip2	61.16	464.h264ref	19.74
403.gcc	1.19	471.omnetpp	7.84
429.mcf	4.66	483.xalancbmk	8.99
445.gobmk	29.60	444.namd	8.08
456.hmmer	4.53	453.povray	6.12
458.sjeng	110.99	470.lbm	18.38
462.libquantum	18.27	482.sphinx3	17.51

memory, in a single-channel configuration, the ORAM overhead is from 2x to 24x, and an average of about 11x. In a 4-channel configuration, the ORAM overhead is from 1.8x to 21x, with an average overhead of about 6.5x.

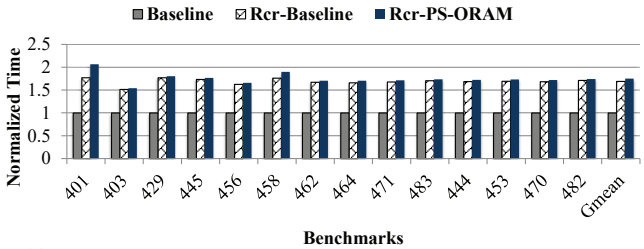
- **FullNVM:** It refers to a system with on-chip stash and PosMap, and off-chip memory built with PCM. It does not support crash-consistent ORAM because the metadata and data blocks are not written back atomically. FullNVM (STT) uses STTRAM to construct on-chip stash and PosMap, and PCM as the main memory.
- **Naïve-PS-ORAM**: It refers to the approach of persisting *all* the accessed data blocks and metadata entries into the ORAM tree and the trusted NVM each time an ORAM access is performed.
- **PS-ORAM**: It refers to the approach of persisting all the accessed data blocks and *dirty* metadata entries into the ORAM tree and the trusted NVM each time an ORAM access is performed.
- **Rcr-Baseline:** It refers to the baseline implementation of the recursive ORAM [19] protocol with NVM. The metadata in PosMap is written back to untrusted NVM in a tree organization every time. Similar to the Baseline without recursion, this scheme does not support data crash consistency.
- **Rcr-PS-ORAM:** It refers to recursive version of PS-ORAM: the metadata in PosMap is written back to untrusted NVM in a tree organization every access. Moreover, the dirty blocks in the stash are persisted for crash recoverability.

5.2 Evaluation Results

In this subsection, we compare the performance and introduced addtional read and write accesses of our proposed designs (see section 5.1 for details) and reported the normalized results to the Baseline (without data persistency).



(a) Performance comparison of different designs in single-channel system.



(b) Performance comparison of PS-Recursive ORAM in single-channel system.

Figure 5: Performance comparison ($Z = 4$, $channel = 1$, $core = 1$).

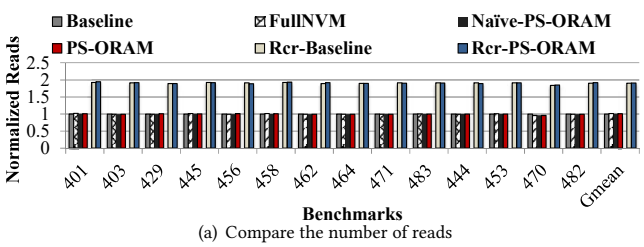
5.2.1 System Performance. Figure 5 shows the normalized execution time in a single channel memory system. Figure 5(a) illustrates the impact of different designs on the performance of non-recursive ORAM systems when performing different workloads. We have the following observations:

a) FullNVM and FullNVM(STT), compared with Baseline, degrades the performance by about 90.54% and 37.69% on average, respectively. Because the read/write latency of STTRAM and PCM is longer than that of SRAM/DRAM, the performance loss is high.

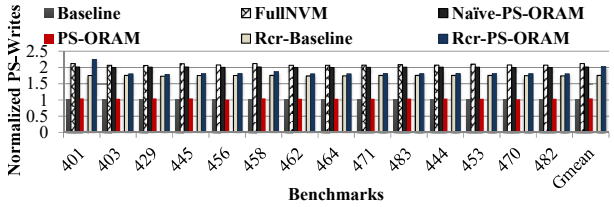
b) Na?ve-PS-ORAM has a slightly better performance than that of FullNVM design. Compared with the Baseline, the average performance is reduced by 73.92%, performance improved by 16.63% over FullNVM. This is mainly because the traditional non-volatile stash on the chip side is faster than NVM to read/write. However, since all the data blocks and metadata entries of one ORAM access need to be persisted, the data persistency overhead is still high.

c) PS-ORAM, compared with the Baseline, the performance loss of PS-ORAM design is only about 4.29%. Compared with FullNVM and Na?ve-PS-ORAM, the performance of PS-ORAM is improved by 86.26% and 69.63%, respectively. This is because the PS-ORAM design only persists dirty metadata entries and the path with accessed data in the write-back path, reducing unnecessary redundant metadata write operations.

Figure 5(b) shows the performance of persistent recursive ORAM (Rcr-PS-ORAM) compared to Rcr-Baseline. Obviously, both the Rcr-Baseline and Rcr-PS-ORAM have a high overhead compared to the non-recursive Baseline. The average performance loss is



(a) Compare the number of reads



(b) Compare the number of writes

Figure 6: Comparison of reads and writes of different designs.

about 68.93% and 75.10%, respectively. However, compared with the performance of Rcr-Baseline, the overhead of Rcr-PS-ORAM is relatively small, about 3.65%. This is because Rcr-Baseline already support PosMap persistency with write backs on every ORAM access, so Rcr-PS-ORAM only needs to provide additional data persistence for the data blocks in the stash.

5.2.2 NVM read/write traffic. Figure 6 shows the comparison of memory read/write traffic between an ORAM system without a persistent design and an ORAM system with a persistent design in a single channel system. From Figure 6(a), we can see that when recursive ORAM executes ORAM read access, compared to Baseline, the number of read accesses increases significantly, the average increase was about 90.28% and 90.54%, respectively. For other evaluated ORAM systems, read accesses remain unchanged. This is because recursive ORAM performs additional path access for reading PosMap entries, resulting in a significant increase in reading traffic accesses.

For the write traffic, from Figure 6(b), we can see that the FullNVM design has the largest persistent write traffic overhead, which is 111.63% more than the Baseline. Since every ORAM access needs to transfer massive data from the NVM-ORAM tree to the on-chip NVM stash and PosMap, the writes to the on-chip NVM is significant.

Other designs have shown similar memory read/write traffic. The PS-ORAM design has the least increment in write traffic, with an average of about 4.84%. Compared with FullNVM and Na?ve-PS-ORAM, the write traffic of PS-ORAM decreased by 106.79% and 96.07%, respectively. As we've discussed, PS-ORAM only write back dirty metadata entries in the PosMap. Compared with the Na?ve-PS-ORAM design, the PS-ORAM design reduces many redundant data persistency operations of PosMap metadata. Compared with the Rcr-Baseline and the Rcr-PS-ORAM design, the write traffic of the Rcr-PS-ORAM design increases, about 15.54% , which is caused by the fact that the Rcr-PS-ORAM design needs to back up the accessed target data blocks every time the execution is a stash eviction.

5.2.3 Multi-Channel Performance. We show how memory bandwidth may affect the performance of each design. By increasing the memory channel number from 1 to 4, we observe better performance for all schemes, as shown in Figure 7. The performance of the PS-ORAM design in the 2-channel and 4-channel settings is 51.26% and 53.76% higher than the performance under the single-channel setting, respectively. The Rcr-PS-ORAM design improved performance by 46.50% and 55.21% in the 2-channel and 4-channel settings over the single-channel Settings, respectively. In the 2-channel and 4-channel settings, the performance of PS-ORAM is lower than that of Baseline by 4.94% and 5.32%, respectively. Similarly, the performance of Rcr-PS-ORAM is lower than Rcr-Baseline by 2.12% and 5.36% respectively.

When the number of memory channels increases to 4, the performance is not significantly improved over 2-channel setting. This is because when the number of channels increases, it is hard to allocate the memory accesses to each channel equally to gain the optimal throughput [63, 64], that is, the relationship between the number of channels and performance is not linear.

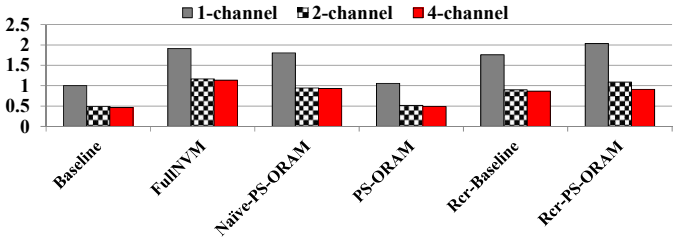


Figure 7: Performance comparison in multi-channel systems.

6 CONCLUSIONS

In this paper, we introduce PS-ORAM, to support efficient crash consistency for general ORAM protocols on NVM. To the best of our knowledge, this is the first work to solve the crash consistency problem of the ORAM system. We first analyze the basic ORAM protocol without data persistency, and find that if the system crashes when performing ORAM access, the data cannot be effectively recovered automatically, which eventually leads to the error of ORAM access. To address the challenge of providing crash consistency support for ORAM, we propose several viable solutions and the best protocol with low overhead. The experimental results show that the proposed data persistency method is not only applicable to traditional ORAM systems, but also to recursive ORAM systems. We believe that our work provides holistic system support for data persistency, crash consistency, and security for future NVM systems.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers for this work in HPCA 2021 and 2022, MICRO 2021 and ISCA 2022 for their constructive feedback.

REFERENCES

[1] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX.. In *NDSS*.

[2] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX.. In *NDSS*.

[3] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. 2021. BBB: Simplifying Persistent Programming using Battery-Backed Buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 111–124.

[4] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1753–1758.

[5] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 707–722.

[6] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. Obfusmem: A low-overhead access obfuscation for trusted memories. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 107–119.

[7] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*. 104–115.

[8] Sundeep Bajikar. 2002. Trusted platform module (tpm) based security on notebook pcs-white paper. *Mobile Platforms Group Intel Corporation* 1 (2002), 20.

[9] Brad Benton. 2017. CCIX, GEN-Z, OpenCAPI: OVERVIEW & COMPARISON. In *13th ANNUAL WORKSHOP 2017*.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[11] Dingyuan Cao, Mingzhe Zhang, Hang Lu, Xiaochun Ye, Dongrui Fan, Yuezhi Che, and Rujia Wang. 2021. Streamline Ring ORAM Accesses through Spatial and Temporal Optimization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 14–25.

[12] Yuezhi Che, Gang Liu, and Rujia Wang. 2021. Seeds of SEED: Efficient Access Pattern Obfuscation for Untrusted Hybrid Memory System. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 63–69.

[13] Yuezhi Che and Rujia Wang. 2020. Multi-Range Supported Oblivious RAM for Efficient Block Data Retrieval. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 369–382.

[14] Yuezhi Che, Yuanzhou Yang, Amro Awad, and Rujia Wang. 2020. A Lightweight Memory Access Pattern Obfuscation Framework for NVM. *IEEE Computer Architecture Letters* 19, 2 (2020), 163–166.

[15] Jianxi Chen, Qingsong Wei, Cheng Chen, and Lingkun Wu. 2013. FSMAC: A file system metadata accelerator with non-volatile memory. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–11.

[16] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, et al. 2012. A 20nm 1.8 V 8Gb PRAM with 40MB/s program bandwidth. In *2012 IEEE International Solid-State Circuits Conference*. IEEE, 46–48.

[17] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[18] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 133–146.

[19] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. 2015. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 103–116.

[20] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. 2015. A low-latency, low-area hardware oblivious RAM controller. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 215–222.

[21] 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.

[22] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 182–194.

[23] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[24] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform storage performance with 3D XPoint technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[25] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[26] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, et al. 2020. DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 385–399.

[27] Weizhe Hua, Zhiru Zhang, and G Edward Suh. 2018. Reverse engineering convolutional neural networks through side-channel information leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[28] Jianming Huang and Yu Hua. 2021. Update the Root of Integrity Tree in Secure Non-Volatile Memory Systems with Low Overhead. *arXiv preprint arXiv:2103.03502* (2021).

[29] Intel. 2020. Intel Optane Technology: Revolutionizing Memory and Storage. Retrieved 2022 from https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[30] intel. 2021. eADR: New Opportunities for Persistent Memory Applications. Retrieved 2022 from https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html

[31] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation.. In *Ndss*, Vol. 20. Citeseer, 12.

[32] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[33] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. Crash consistency validation made easy. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 133–143.

[34] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel® software guard extensions: Epid provisioning and attestation services. *White Paper* 1 (2016), 1–10.

[35] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[36] Kimberly Keeton. 2015. The machine: An architecture for memory-centric computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, Vol. 10.

[37] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An off-chip attack on hardware enclaves via the memory bus. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*.

[38] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *Acm Sigplan Notices* 35, 11 (2000), 168–177.

[39] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. 2014. NVM Duet: Unified working memory and persistent store architecture. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 455–470.

[40] Sihang Liu, Aasheesh Kolli, Jinglei Ren, and Samira Khan. 2018. Crash consistency in encrypted non-volatile main memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 310–323.

[41] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. Phantom: Practical oblivious computation in a secure processor. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 311–324.

[42] Dhinakaran Pandiyan and Carole-Jean Wu. 2014. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 171–180.

[43] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 433–448.

[44] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[45] Matthew Poremba, Tao Zhang, and Yuan Xie. 2015. Nvmain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems. *IEEE Computer Architecture Letters* 14, 2 (2015), 140–143.

[46] Joydeep Rakshit and Kartik Mohanram. 2018. LEO: Low overhead encryption ORAM for non-volatile memories. *IEEE Computer Architecture Letters* 17, 2 (2018), 100–104.

[47] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. 2015. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 672–685.

[48] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements

to Oblivious RAM.. In *USENIX Security Symposium*. 415–430.

[49] Ling Ren, Christopher W Fletcher, Xiangyao Yu, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2014. Unified Oblivious-RAM: Improving Recursive ORAM with Locality and Pseudorandomness. *IACR Cryptol. ePrint Arch.* 2014 (2014), 205.

[50] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. 2013. Design space exploration and optimization of path oblivious ram in secure processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 571–582.

[51] Andy M Rudoff. 2016. Deprecating the pcommit instruction. Retrieved 2021 from https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html

[52] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–217.

[53] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. 2017. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *IACR Cryptology ePrint Archive* 2017 (2017), 549.

[54] Steve Scargall. 2020. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature.

[55] Priya Sehgal, Sourav Basu, Kiran Srinivasan, and Kaladhar Voruganti. 2015. An empirical study of file systems on nvm. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–14.

[56] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, et al. 2013. RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 185–197.

[57] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, Todd C Mowry, and Trishul Chilimbi. 2015. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 79–91.

[58] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 299–310.

[59] Shivam Swami, Joydeep Rakshit, and Kartik Mohanram. 2016. SECRET: Smartly encrypted energy efficient non-volatile memories. In *Proceedings of the 53rd Annual Design Automation Conference*. 1–6.

[60] Shruti Tople, Yaoqi Jia, and Prateek Saxena. 2019. Pro-oram: Practical read-only oblivious {RAM}. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 197–211.

[61] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *FAST*, Vol. 11. 61–75.

[62] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[63] Rujia Wang, Youtao Zhang, and Jun Yang. 2017. Cooperative path-oram for effective memory bandwidth sharing in server settings. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 325–336.

[64] Rujia Wang, Youtao Zhang, and Jun Yang. 2018. D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 416–427.

[65] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. 2016. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. 1–7.

[66] Fan Yang, Youyou Lu, Youmin Chen, Haiyu Mao, and Jiwu Shu. 2019. No compromises: Secure NVM with crash consistency, write-efficiency and high-performance. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[67] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 167–181.

[68] 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

[69] Vinson Young, Prashant J Nair, and Moinuddin K Qureshi. 2015. DEUCE: Write-efficient encryption for non-volatile memories. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 33–44.

[70] Xian Zhang, Guangyu Sun, Peichen Xie, Chao Zhang, Yannan Liu, Lingxiao Wei, Qiang Xu, and Chun Jason Xue. 2018. Shadow block: accelerating ORAM accesses with data duplication. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 961–973.

[71] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. 2015. Fork path: improving efficiency of oram by removing redundant memory accesses. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 102–114.

[72] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2004. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. *ACM SIGOPS Operating Systems Review* 38, 5 (2004), 72–84.

[73] Pengfei Zuo, Yu Hua, and Yuan Xie. 2019. SuperMem: Enabling application-transparent secure persistent memory with low overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 479–492.