TOWARDS EFFICIENT SECURE MEMORY SYSTEMS WITH OBLIVIOUS RAM

by

Rujia Wang

B.E., Zhejiang University, 2013

M.S., University of Pittsburgh, 2015

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2018

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Rujia Wang

It was defended on

May 24, 2018

and approved by

Jun Yang, Ph.D., Professor, Department of Electrical & Computer Engineering

Youtao Zhang, Ph.D., Associate Professor, Computer Science Department

Kartik Mohanram, Ph.D., Associate Professor, Department of Electrical & Computer Engineering

Samuel Dickerson, Ph.D., Assistant Professor, Department of Electrical & Computer Engineering

Natasa Miskov-Zivanov, Ph.D., Assistant Professor, Department of Electrical & Computer Engineering

Adam J. Lee, Ph.D., Associate Professor, Computer Science Department

Dissertation Director: Jun Yang, Ph.D., Professor, Department of Electrical & Computer Engineering

TOWARDS EFFICIENT SECURE MEMORY SYSTEMS WITH OBLIVIOUS RAM

Rujia Wang, PhD

University of Pittsburgh, 2018

When multiple users and applications share the resources on cloud servers, information may be leaked through hidden channels related to the memory. Encryption can help to protect data privacy. However, the physical address on the memory bus cannot be encrypted if there is no computation power on memory DIMM. The attacker may observe clear-text physical address access frequency and infer sensitive information in the program. To completely protect the system from address access pattern leakage, we need to use Oblivious RAM, which obfuscates the physical address by remapping it after each access. However, the ORAM access is still costly regarding bandwidth.

In this dissertation, I focus on discussing and designing efficient and scalable secure memory systems with ORAM. Firstly, I studied the co-run interference between different applications on the modern computer servers. We found out that how to allocate shared resources between secure applications and other normal applications will determine the overall system performance. I proposed Cooperative-ORAM protocol, which achieves the goal of better resource allocation, utilization and same security guarantee as original ORAM design. Our design delivers an average of 20% overall performance improvement over the baseline Path ORAM design while providing a flexible resource tuning between different kinds of applications.

In the next part, I address the problems when the application number further scales on the same server. The co-run interference and memory traffic will be more intense when we scale the number of applications on the server. Meanwhile, more applications mean that the

demand for memory capacity is also increasing. I proposed the design of D-ORAM, which delegate the ORAM based secure engine on Buffer-on-Board(BoB), which is in between of the last level cache and main memory, to enable high-level privacy protection and low execution interference on cloud servers. By pushing the ORAM engine off-chip, most of the ORAM accesses will not need to be sent back to the processor side, which removes the excessive data movement overhead. Our evaluation shows that D-ORAM improves normal applications performance by 22.5% on average.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

There are many memorable and meaningful moments during my journey to a Ph.D. degree. I can still clearly remember the bitterness when my first draft turned out a failure and the sweetness of my first acceptance. The way to a Ph.D. degree is never easy and always full of challenges, but thanks to everyone I have met and worked with during these years, I finally make my way to it.

Firstly, I would like to thank my advisors, Prof. Jun Yang and Prof. Youtao Zhang. Being their student is one of the luckiest things that happened in my life. They not only spend a lot of time helping me explore the new area, but also give comprehensive advice about my future career, which will be helpful throughout my entire life. I would like to thank all my committee members, Prof. Mohanram, Prof. Dickerson, Prof. Miskov-Zivanov and Prof. Lee, for their time and constructive comments on the dissertation proposal and defense.

Next, I also want to thank all of my group members and all great researchers I have met during my travel to conferences, internships, and job search. Sharing and learning knowledge from each other helps me to learn knowledge comprehensively. I would like to sincerely thank my family, who are always supporting and encouraging me during my study. To my parents, who give me the freedom and encouragement to choose my life and pursue my dream, and to my husband, my most significant life companion who teaches me to be independent while backing me up all the time.

Lastly, I would like to thank myself, being continuously hardworking and optimistic. To infinity and beyond!

1.0 INTRODUCTION

Cloud computing is becoming extremely popular these days. It offers the opportunity to share powerful computing resources, while still provide different users with private computation and storage. By upgrading the hardware on the cloud infrastructures, such as adopting secure processors, can enable secure computation on the cloud. In such setting, the user sends encrypted data to the cloud, and the secure processor can decrypt data and do the computation. After the computation, data need to re-encrypt and send back to the user. Many secure processors and platform have been proposed, including XOM [44] and TPM [35].

Based on the secure processor, we can assume that the computation is tamper-resistant. However, in the communication between secure processor and memory, attackers can still observe information leakage through side channels. Preventing information leakage on side channel requires extra protection. We discuss two types potential information leakage, access pattern leakage and timing channel leakage. Access pattern leakage refers to that an adversary can observe the location program accessed to infer sensitive information. For example, by tampering with the memory address bus, an adversary can extract important information from the observed access patterns. For example, the files being accessed in a cloud file system [84, 70], the disease/specialist information being looked up in a medical application [11], and the queries being executed on a database [2], may all leak sensitive information.

To completely preventing access pattern leakage, we need to use Oblivious RAM [27, 28], which randomly assigns a new address to the data block accessed, or shuffles the RAM periodically. The concept of ORAM was first introduced by Goldreich and Ostrovsky [27, 28], and more and more follow-up ORAM algorithms and optimizations have been proposed in recent years[96, 61, 72, 59].

The recent advance on ORAM proposed Path ORAM [72], a simple and practical ORAM protocol that has greatly improved memory bandwidth. Although Path ORAM reduces memory bandwidth usage from O(N) to O(log N), one memory access is still translated to tens actual DRAM read and write operation. Ring ORAM was proposed to[59] further reduce the theoretical memory bandwidth to constant, however, the algorithm introduces more sophisticated control overhead.

To efficient utilize secure ORAM algorithms on the cloud, we need to consider how ORAM operations are performed on existing DRAM systems. In this dissertation, I will analyze these algorithms with emphasis on improving memory bandwidth sharing and utilization.

1.1 MAJOR CHALLENGES

Although the algorithms of ORAM has been improved over the years, most of the performance and overhead analysis is still based on the assumption that all memory accesses are equal. Moreover, the improvement on algorithm side may always introduce other control complexity on the hardware side. Also, the architecture details of the memory system are not taken into consideration. Therefore, the maximum throughput a system providing is not fully utilized with default scheduling and allocation techniques.

Besides, most of the state-of-art ORAM optimization techniques only consider the single secure application itself. The computing model assumes a single ORAM application is taking over the entire computing resources. However, in most cases, especially with the evolution of cloud computing and services, multiple applications or virtual machines share same physical hardware. Therefore, the current computing model is too simple to meet the complex system. A more reasonable computing model which contains both secure and normal applications should be considered when the sharing becomes pervasive.

Moreover, the potential of using new memory architecture to accelerate ORAM application is not well studied yet. Conventional DRAM architecture has limited computation power on the memory side, which requires all memory accesses introduced by ORAM must be processed inside of the processor. There are several potential ways to add computing logic

on or closer to memory side, which brings a tremendous opportunity to reduce ORAM overhead. Meanwhile, the trade-offs and security guarantee modifications need to be discussed for proposed new architecture accelerating the ORAM applications.

In this dissertation, I study the ORAM application memory behavior in micro-command level, and characterize it as memory intensive but with high locality, which means that a special memory scheduling for it need to be implemented in the memory controller. I study the co-run scenario and interference with multiple applications and find out the wasted resources with the new computing model. I also explore designs to remove the co-run interference entirely without sacrificing architecture parallelism with new memory architecture.

1.2 RESEARCH OVERVIEW

This dissertation targets at building an efficient, secure memory system with ORAM. It covers several works that are aiming at improving ORAM applications performance on a shared server environment. More specifically, we studied on the micro-architectural level of current memory systems and proposed memory scheduling techniques to enhance system utilization rate and remove co-run interference. Our work does not change ORAM algorithm, which guarantees the same security protection as an original system while improving the performance of different types of applications. The outline of the dissertation is shown below.

In Chapter 2, I introduce the background knowledge on memory security, types of potential attacks on memory side, memory information leakage, state of the art ORAM workflow, and memory organizations. In addition, an overview of current secure memory systems is given and discussed.

In Chapter 3, I introduce the first work, CP-ORAM, which was published in HPCA 2017. This work targets at investigating the cause of slow down on a shared server, when consolidating different types of applications sharing the memory bandwidth. We have proposed software and hardware co-design techniques that can allow applications improve the memory bandwidth utilization rate.

In Chapter 4, I introduce the second work, D-ORAM, which further reduce the interference between applications by an ORAM delegator design. Our approaches enable low execution interference when we continue to scale the number of applications run on the server, which puts higher pressure on the secure memory system. Our flexible capacity expansion technique can help the user to build a more scalable ORAM system with low overhead.

In Chapter 5, I discuss selected related works in this field that are highly related to our approaches. Chapter 6 includes several potential future projects and challenges that are related to this dissertation. The final Chapter 7 concludes the dissertation and the main contributions.

2.0 BACKGROUND

2.1 MEMORY ARCHITECTURE, ORGANIZATION AND INTERFACE

DRAM is widely used as main memory in current computing systems. To increase the parallelism of the DRAM system, it is organized as a hierarchy of channels, ranks, banks, and arrays[36]. Each channel of the system is independently connected to the on-chip memory controller. Inside of each channel, one or multiple DIMMs can provide the capacity for such channel, and each physical DIMM contains one or more ranks. Ranks in the same channel will share global bus and bandwidth connecting to the on-chip memory controller. Inside of each rank, there are multiple banks (typically 8) that can provide internal parallel access capability, which is also called bank-level parallelism. Bank is the smallest structure inside of the DRAM that can be operated in parallel without hardware modification.

The DRAM arrays inside of each bank are arrays of DRAM cells, which is composed of a transistor and a capacitor. The charge inside the capacitor indicates whether this cell store a '0' or '1'.

To operate the DRAM from high level, the memory access commands need to go through the memory controller and being translated into standard operations via DRAM interface. There are various standard DRAM interfaces available in current market. Among them, the mainstream protocol standard defined by JEDEC is Double Data Rate(DDR), which transfer data on a dual rate memory bus. The variations of DDR, such as DDR3[68], DDR4[3], differs from the bus clock and data transfer rate.

To read or write a line from the DRAM, the memory controller needs to issue a read or write request, and it will be decomposed into micro commands that DRAM can handle. When a read request arrives at the memory controller, a precharge is issued if the row is not

previously read into the row buffer. Then an activation command is issued, after the timing constraint is met, a read command will be sent, and data will be sent out via the data bus. The JEDEC specify the minimum timing constraints between each command to the DRAM, and also maximum parallel commands a DRAM module can operate[36].

Except for the direct attached conventional DRAM modules, High Bandwidth Memory(HBM), is one of the most promising stacked memory technology used for high-performance graphics computing[69]. It has been adopted by JEDEC as a standard memory interface since 2013. The interface is similar to DDR, especially for the data and column/row command and address bus. Other updates of the HBM interface include split command interface, the single bank refresh, and RAS support.

Hybrid Memory Cube(HMC), which was released in 2013, is another type of stacked DRAM with high memory bandwidth[13, 12]. It uses standard DRAM cells but has more banks than the conventional DRAM modules. HMC combines TSVs and microbumps to multiple DRAM dies, with a logic layer on the bottom as a separate die. The interface of HMC is different from DDR, and it requires a packetize engine to send data and address together via a serial link.

2.2 ATTACKS ON MEMORY SYSTEM

In this section, I summarize the major types of memory attacks that are popular on commodity computing systems. Various of attacks that can obtain unprivileged data have been proposed and proved on the different levels of the memory system.

2.2.1 Cache attacks

Cache attacks, which are the most repeatable type of side-channel attacks, can leak sensitive information through the timing difference between a cache hit and miss. Most of the cache attacks focus on shared last level cache, as it can be utilized by multiple processes or multiple VMs on the cloud server.

A typical setting of cache attack requires two types of process sharing the cache set. The victim process is executing algorithms that contain sensitive information such as encryption key, and the spy process can alter the cache states and affect the execution of victim process. Three representative cache attacks are described as below.

Evict and time. [55] The spy is able to evict any cache line shared with victim process. Then, the spy process time the victim's execution time before and after evicting a cache set. If the time of victim function after the eviction is much longer than before, it means that the victim process occupies the address of such evicted set.

Prime and Probe. [46] In this attack, the spy will first prime a cache set with a specific memory address. Then it will wait for the victim activity on the same cache set. Later, the spy access the original cache set with the memory address again and time the difference. If the second spy access is slower than the first one, then it means the victim has used this address and replaced the data block within the set.

Flush and reload. [89] The spy uses instruction such as *cflush* to flush a line in the shared cache, which leads to a cache miss for both applications. After the victim's execution, the spy time the access for accessing the same shared address. If the second time is faster than the first one, it means that the block is accessed and brought back by the victim.

2.2.2 Main memory attacks

Cold boot attack. The DRAMs in the system use charge to store data, therefore, they are expected to lose all the content when the system is powered off. However, studies are showing that data stored in the DRAM can still retain for a period [47, 31]. In such case, DRAM cannot be seen as volatile anymore, and it will raise up potential security risks as data are not completely destroyed as expected.

Cold boot attack[31, 66, 90] exploits the non-destructive data in DRAM when the system is powered off or reboot. The researchers are able to [31] recover the disk encryption keys from DDR and DDR2 DRAMs by cooling the DRAM and transferring the DIMMs to an attackers machine. Till recent years, researchers are still able to construct cold boot attacks using most recent DDR4 DRAM with memory scramblers[90].

Replay attack. During data transmission, an attacker can intercept the data and maliciously re-transmits it. This type of attack is called replay attack, and it can happen on the network, as well as on the memory bus in the computing system.

Specifically, a memory block with a given address can be recorded by the attacker in the middle, and been inserted with the same address at a later point in time. By doing so, the current blocks value is replaced by an older one, but still a valid one. Such an attack may be viewed as a temporal permutation of a memory block, for a specific memory location. Protecting data from replay attack requires integrity tree and MACs check on every access [64, 1].

2.3 MEMORY CHANNEL INFORMATION LEAKAGE

In this section, I will discuss two types of memory channel information leakage, which will harm the privacy of data stored in the main memory. By tampering the memory bus, the attacker can observe the access pattern, i.e., physical address location, and timing between consecutive memory accesses. Even with data encryption, information can still be leaked through the memory I/O.

2.3.1 Access pattern leakage

Access pattern means the physical address location a program may access during the execution. Figure 1 shows a branch that may leak the information. By looking at the memory address accessed, the attacker can infer whether the sensitive variable key is 1 or not. Sequentially access to memory will tell the attacker that the key is not 1, while access to the same location with address 0 will lead to the conclusion that key is 1.

Therefore, to prevent access pattern leakage, even if we always want data content in address 0, we should access different location of the memory. This is the key idea of ORAM, which after each access, the physical address need to be reshuffled to a different location.

```
for i=1 to n
        if(key[i] == 1)
                sum+= mem[0]
        else
                sum+= mem[i*4]
```

Figure 1: A program that leaks through access pattern.

2.3.2 Timing side channel leakage

Timing side channel leakage refers to that an attacker can observe the access rate and infer important parameters in a program. By observing the access rate on the memory bus, the attacker may get knowledge of a sensitive variable's value [24]. One representative example is the RSA decryption algorithm, uses a private key to decrypt an encrypted message. It is often implemented with the square and multiplies algorithm to perform fast exponentiation, and the bits in the private key are checked one by one, and a modulo operation is performed when the bit is 1.

A more straightforward example is shown in Figure 2. In this attack example, the memory bus will be busy if the secret key bit is "0". Therefore, by observing the memory traffic rate on the bus, the attacker can infer sensitive key information.

```
for i=1 to n
        if(key[i] == 1)
                sum++
        else
                sum+= mem[i*4]
```

Figure 2: A program that leaks through timing channel.

To prevent the timing channel attack, the processor can issue memory request on a fixed rate. If there is no memory request at an issue point, a dummy request will be issued instead. In recent years, researchers have proposed several efficient scheduling to find optimum memory request issue rate against timing channel attack[82, 83].

2.4 OBLIVIOUS RAM

Oblivious RAM (ORAM) [27, 28] is a cryptographic primitive for preventing information leakage from memory access patterns. ORAM conceals the access pattern from an application by continuously shuffling and re-encrypting the memory data after each access. An adversary, while still being able to observe all the memory addresses transmitted on the bus, has a negligible probability to extract or distinguish the real access pattern, which ensures that the honest but curious server cannot learn any information through client or application access pattern on the address bus.

The ORAM protocol ensures the translation from virtual address to physical address is oblivious. Here, the virtual address refers to the user's request address, indicating which data block the application needs to read or write. The ORAM protocol translates each virtual address into a sequence of actual physical addresses, and send it to the memory controller to perform the bulk of read and write operations.

The basic implementation of ORAM requires reshuffle and re-encrypt all memory data in a memory on every access. This means that whenever there is a read or a write to the memory, the entire content needs to be read out, decrypt, and after the desired block is found, all contents need to be re-encrypt and written back to the memory. Consider a memory space of N, the reshuffle process imposes an $O(N)$ overhead, makes it only stays in the theoretical level to protect a system from access pattern, as nowadays, the main memory capacity is in the range of GB. To further reduce the access overhead, various types of efficient ORAM schemes have been proposed.

2.4.1 Partition based ORAM

Partition based ORAM[71] divides one large ORAM space into multiple partitions so that each time, only a portion of the memory needs to be reshuffled. Consider a memory space which contains N data blocks, each partition contains $\sqrt{N}$ blocks, and there are total $P = \sqrt{N}$ partitions in total.

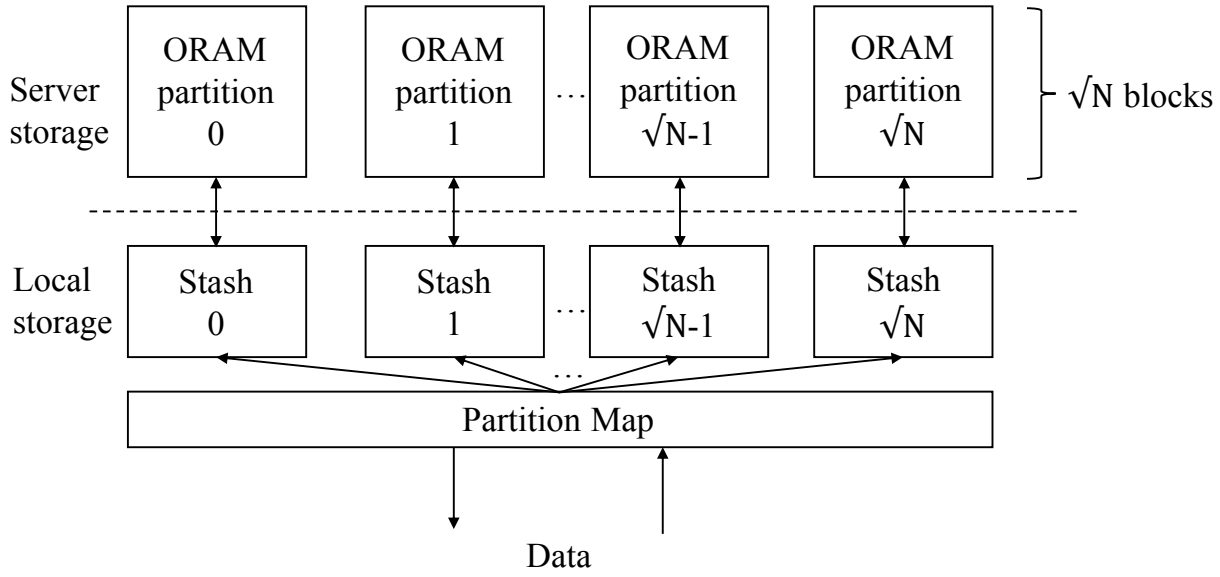


Figure 3: The Partition ORAM scheme.

On every memory access, the partition ORAM finds the partition that stores the block first. Each ORAM partition can be operated with different ORAM schemes. To ensure the obliviousness, the same block needs to be remapped into a different random partition. In this way, the server is only able to observe which partition is being accessed but does not learn any operations inside of the partition, such as which location stores the block.

On the client side, each partition needs to keep a stash, which temporarily buffers a block that just been mapped to the partition. Partition map on the client side keeps track of where a data block is mapped. After each partition read operation is done, the content inside of the stash needs to be written back to the partition. To ensure the obliviousness, each stash selects v blocks and write them back to the server side ORAM partitions. Dummy blocks can be appended when there are not enough blocks in the stash.

Figure 3 shows the overview of Partition ORAM architecture. On the worst case when original ORAM is used, the access overhead is $O(\sqrt{N})$. The client-side storage overhead is $O(N)$, since the partition map needs to record the mapping for each block. This overhead can be reduced by recursive construction.

2.4.2 Tree based ORAM

To further reduce the access overhead, a tree-based organization is proposed and adopted in many ORAM constructions. The major similarity for all tree-based ORAM is that the main memory on the server side is organized into a tree and each node of the tree contains several data blocks. Compared to the partition ORAM, tree-based ORAM assigns a new leaf on every access. Path ORAM and Ring ORAM are two representative ORAMs in this category.

Path ORAM [72] was recently proposed as a practical ORAM implementation. In Path ORAM, the unsafe memory is structured as a balanced binary tree, where each node is referred to as a bucket that can hold `Z` blocks (a block is often of the size of a cache line, i.e., 64B). The tree has `L+1` levels — the root of tree is at level `0` while the leaves are at level `L`. The total number of data blocks that an ORAM tree can hold is $\texttt{N=Z*(2}^{\texttt{L+1}}\texttt{-1)}$. For the example in Figure 4, we have `L=3`, `Z=4`, and `N=60`.

The blocks in the Path ORAM tree can be either real data blocks or dummy blocks. The dummy blocks are introduced as space filler, which may be replaced with real data blocks if needed. An adversary cannot differentiate dummy blocks from real ones as encryption hides the contents of the blocks.

The ORAM interface for Path ORAM consists of a stash, a position map, the address logic and encryption/decryption logic. The stash is a small buffer that stores up to `C` data blocks from the ORAM tree. The position map is a lookup table that maps program addresses to data blocks in the tree. A Path ORAM tree has $\texttt{2}^{\texttt{L}}$ paths from the root to different leaves. Given an LLC (last level cache) miss, the program address is first sent to the position map to determine on which path the requested block is stored. Assuming the block is on path `l`, the address logic determines the actual DRAM physical address using a static map-

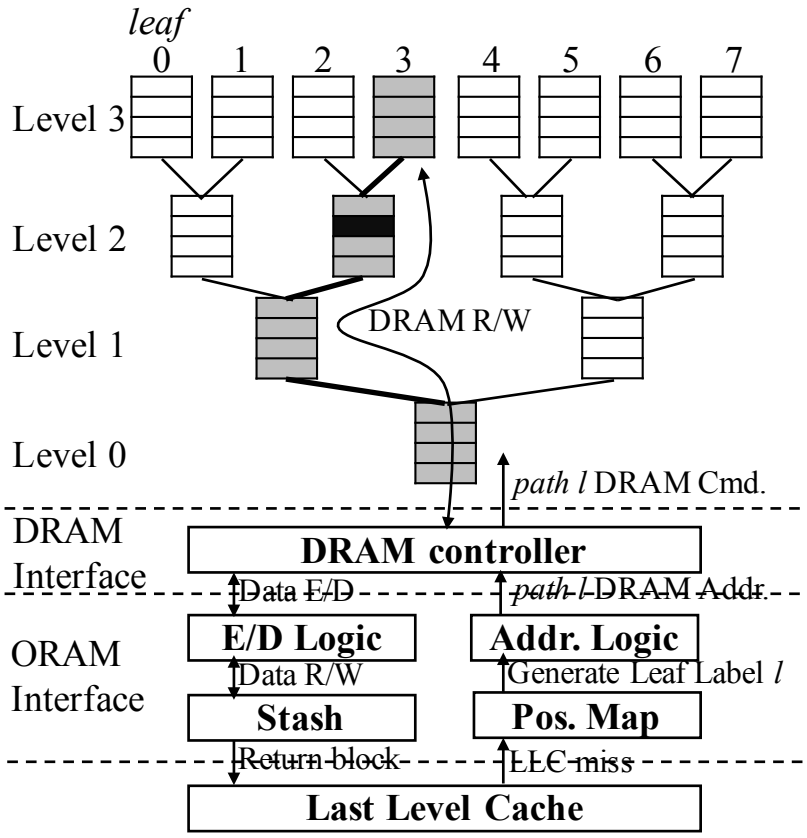


Figure 4: The Path ORAM scheme.

ping table. Then the physical address is sent to DRAM controller and translated to DRAM device commands, such as `PRE`, `ACT`, `RD`, and `WR`, to perform actual memory operations.

Accessing a memory data block starts with the search for the block in the stash. A stash hit terminates the search and returns the block while a stash miss results in a Path ORAM access to the unsafe memory. Assuming that the block is on path `l`, each ORAM access consists of two phases — read phase and write phase.

- In the read phase, all the data blocks on path `l` are read and decrypted and stored in the stash. For an LLC read miss, the requested block is then returned. For an LLC write miss, the requested block gets updated in the stash. After the access, Path ORAM maps the requested block to a new path `l'` while all other fetched blocks are still associated with their original paths.
- In the write phase, data blocks are encrypted with new keys and written to the buckets along path `l`. These buckets are greedily filled in with blocks from the stash. Path ORAM uses the order from the leaf to the root with blocks pushed as many as possible down to the leaf.

  The write phase may write back a block from another path if (i) this block is currently in the stash; (ii) the target bucket is on the overlapped portion of the two paths; and (iii) there is free space in the target bucket.

Path ORAM changes the block-to-path mapping after each access such that the memory accesses from the user application, even if being very regular, are randomized, which effectively prevents information leakage. Path ORAM further eliminates access temporal pattern by issuing accesses at a fixed rate, even if there are no actual accesses [50, 21].

Ring ORAM [59] organizes the memory as a binary tree structure, similar to Path ORAM. Each node in the binary tree is a bucket that can hold a small number of memory blocks. Every block is mapped randomly in the tree leaf bucket, and the remapping information is stored in position map. If a block a is mapped to leaf l by the position map, the block a is stored either along the path from the root to leaf l, or in the stash.

The major difference of Ring ORAM from Path ORAM is the bucket organization. Each bucket has `Z+S` slots and a small number of meta data. In these slots, up to `Z` slots of real

blocks can be stored. At least of `S` dummy blocks must be stored in each bucket. Figure5 shows the different bucket organization of Ring ORAM. In this example, blocks in grey are dummy blocks, and blocks in white are real blocks. There are additional metadata needed such as *count*, *valids*, and *ptrs*. For every bucket in the tree, the physical positions of the `Z+S` real and dummy blocks in each bucket are permuted randomly concerning all past and future writes to that bucket.

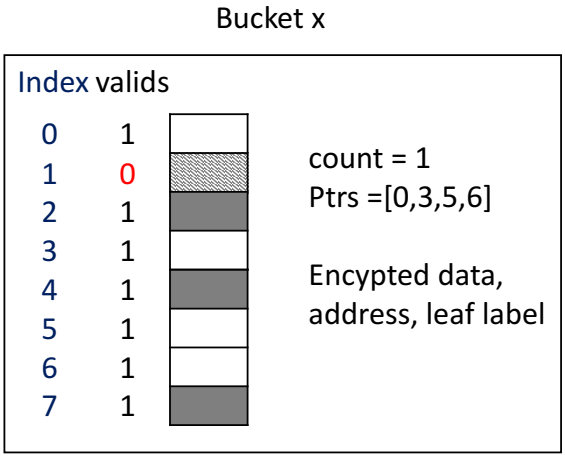


Figure 5: The Ring ORAM Bucket with `Z`=4, `S`=4

The detailed workflow of Ring ORAM is illustrated as follow. There are three basic operations in Ring ORAM, *Read Path*, *Evict Path*, and *Early Reshuffle*.

- *Read Path* operation reads all bucket along one path to look for the block of interest. The bucket that contains the block of interest will return it, while all other buckets along this path will return a random dummy block. Different from Path ORAM read path operation; now each bucket only needs to return one block instead of all blocks. The blocks accessed in each bucket is marked invalid. Therefore, it will not be reaccessed in later access. This lower the online read bandwidth to `L+1` blocks.
- *Evict Path* operation reads and writes a path after every `A` read path operation. It will read all `Z` real blocks in the path and write `Z+S` blocks to the buckets along the path. The write phase will push blocks in stash back to a path. The sole purpose of an eviction operation is to push blocks back to the binary tree from the stash.

- *Early Reshuffle* is needed to maintain each bucket is properly randomly shuffled. Due to randomness, each bucket can only be touched at most `S` times, because after `S` access to a bucket, all dummy blocks will be invalidated. Early reshuffle operation will read and write buckets that have been accessed `S` times and reset the access counter. In Figure 5, the dummy block 1 is accessed and marked as invalid. The counter is set to 1 after this access. When the counter is equal to `S`, which is 4 in the example, the bucket needs to be reshuffled before it can be accessed again.

3.0 COOPERATIVE PATH ORAM FOR EFFECTIVE MEMORY BANDWIDTH SHARING

3.1 PROBLEM STATEMENT

With the fast adoption of cloud computing paradigm, it becomes increasingly important to prevent information leakage from programs running on untrusted cloud servers. Secure processor designs, e.g., XOM [44] and TPM [35], can encrypt and secure the program code, the user data and its execution flow. However, sensitive information may still be extracted through memory access sequences [96, 61]. Studies showed that completely stopping information leakage from memory access patterns requires ORAM (Oblivious RAM) [27, 28], a cryptographic primitive that often incurs large performance overhead. The recent advance on ORAM proposed Path ORAM [72], a simple and practical ORAM protocol that has greatly improved asymptotic efficiency.

Cloud service providers often consolidate multiple applications on one physical server to reduce power and energy consumption, and to maximize system resource utilization. This is also preferred when executing a secure application that adopts Path ORAM — Path ORAM converts each memory access from the secure application to tens to hundreds of memory accesses, which would leave most system resources idle if the secure application monopolizes the server. Therefore, it is natural to consolidate one secure application with one or multiple non-secure applications on one physical server. Unfortunately, due to the extreme memory access intensity in Path ORAM, it is challenging to effectively schedule memory requests from both types of applications. In particular, Path ORAM synchronously distributes its memory accesses across all memory channels, while non-secure applications have much lower access intensity and memory requests exhibit significant imbalance at the channel level. Adopting

traditional memory scheduling schemes does not consider the biased memory traffic of ORAM applications and often wastes large memory bandwidth, which introduces large performance degradation to both types of application.

3.2 THREAT MODEL

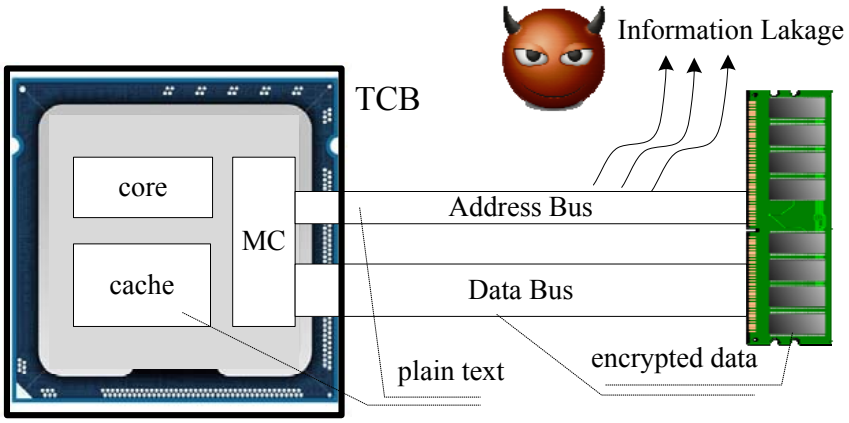


Figure 6: The threat model.

The threat model adopted in this work follows that in previous work [23, 61, 95]. With the processor being the only hardware component within the trusted computing base (TCB) [35], an adversary can access the data stored in main memory and the data communicated on address and data buses. Such access includes potential physical access to the hardware and may be enhanced by specially designed devices and tools, e.g., bus traffic analyzer [76].

To ensure security, the data in the main memory are encrypted with architectural assisted security enhancements [73, 86], which not only ensure data secrecy and integrity but also minimize performance overhead to the system. However, as shown in Figure 6, accessing user data needs to have plain text physical addresses sent to the memory modules, making encryption alone designs insufficient. For example, by tampering with the memory address bus, an adversary can extract important information from the observed access patterns, e.g., the files being accessed in a cloud file system [84, 70], the disease/specialist information

being looked up in a medical application [11], and the queries being executed on a database [2]. Even when both code and data are unknown to the adversary, previous work has demonstrated a control flow graph (CFG) fingerprinting technique to identify known pieces of code solely based on the address trace [96].

The security focus in this work is on preventing information leakage from address access patterns.

3.3 BASELINE SERVER SETTINGS

With fast technology scaling, modern computer servers usually have abundant system resources. For example, the server in Figure 7 adopts a chip-multiprocessor that supports the concurrent execution of multiple threads, and four memory channels each of which can transmit data at 12.8GB/s (DDR3-1600). Cloud service providers often consolidate multiple applications on one physical server to reduce power and energy consumption, and to maximize system resource utilization.

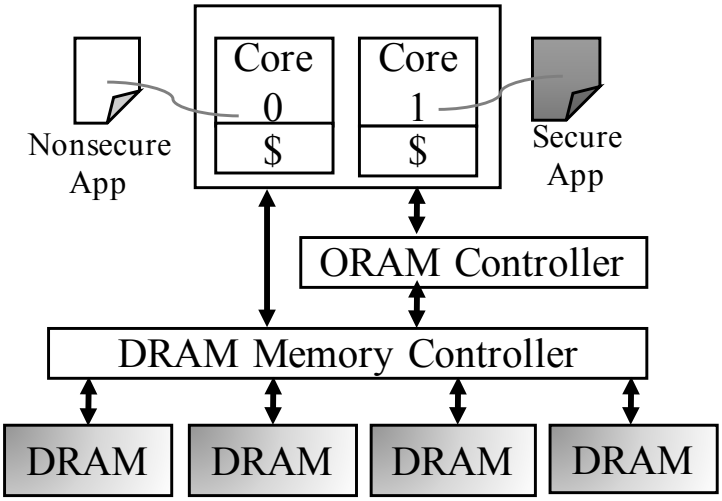


Figure 7: The baseline server setting.

In this work, the baseline secure processor uses two cores — one is dedicated to execute secure application that demands data encryption to protect data secrecy and Path ORAM

to prevent information leakage from access patterns; the other is to execute one non-secure application without data encryption and Path ORAM. These two types of applications are referred to as **S-App** and **NS-App**, respectively. We evaluate the setting with more cores running more than one S-App and NS-App applications in the experimental section. To simplify discussion, we assume that each core has private cache. In the case if a shared last level cache is used, it demands additional security enhancements to prevent potential timing channels and side channels between two applications [82, 20, 46, 45].

The S-App and NS-App applications share the main memory through address partition — for simplicity and without considering the space occupied by the OS, each application may take half of the space of each bank from each channel. There is no physical address overlap between two applications. As a comparison, an alternative design is to share the memory through channel partition — S-App uses a subset of channels while NS-App uses the rest. As we will show in the next section, channel partition is sub-optimal because the channels allocated to NS-App tend to be under-utilized while those allocated to S-App tend to be overloaded.

In this work, we assume that the ORAM memory space is 4GB and each bucket stores 4 blocks such that the tree has 24 levels (`L=23`). We adopt sub-tree layout to spread ORAM memory accesses across all four memory channels. We also adopt tree top caching that cache top 10 levels in a 256KB cache ($4 * 64 * (2^{10} - 1) \approx 256KB$), which eliminates around 42% accesses to the memory. We observe negligible path overlap beyond top 10 levels from consecutive path accesses [95].

3.4 MOTIVATION AND DEGRADATION ANALYSIS

In this section, we first motivate the CP-ORAM design and then develop three schemes to improve memory bandwidth utilization in server settings.

3.4.1 Motivation

The Path ORAM is an extremely memory intensive protocol — [61] showed that a secure application that adopts Path ORAM (i.e., S-App) consumes almost all of the system peak memory bandwidth and introduces 10-100× slowdown over the native execution without adopting ORAM. Consequently, the processor shall be left mostly idle if S-App monopolizes the server. Non-secure applications (i.e., NS-Apps), even those that are traditionally categorized as memory intensive, have much low memory access intensity. Therefore, it is natural to consolidate S-App and NS-App applications on one server to improve resource utilization and save energy consumption.

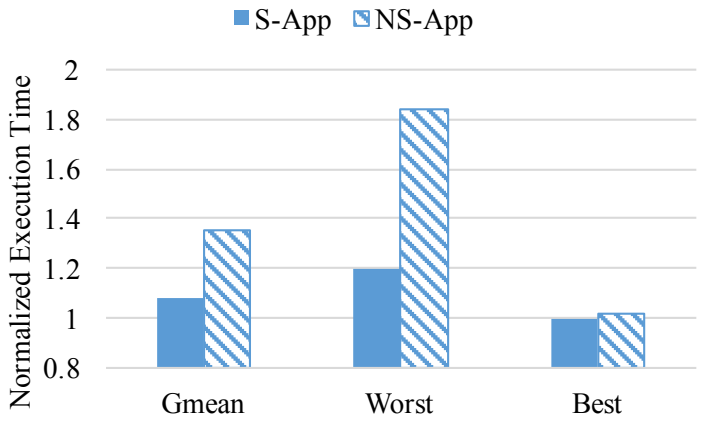


Figure 8: The S-App and NS-App co-run leads to large performance degradation.

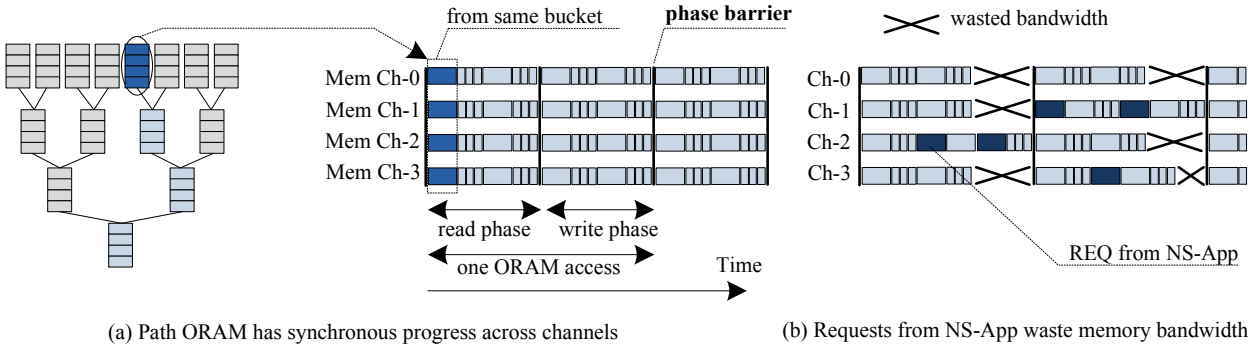
We study the performance degradation when two applications, one S-App and one NS-App, co-run on one server. Figure 8 reports the average (`Gmean`), worst (`Worst`), and best (`Best`) results, which are normalized to the solo execution of each application. `Gmean` averages the results of a suite of workloads. The worst (and best) case considers the summed degradation percentages in different co-runs. The system settings are listed in Section 3.6. From the figure, we have two observations.

- In `Best`, both types of applications have little performance degradation. This indicates that consolidation has the potential to significantly improve resource utilization in the server setting.

- In `Worst`, both types of applications suffers from large degradation. NS-App suffers more percentage degradation than S-App does, i.e., 80% vs 20% degradation when comparing to the solo run.

  While S-App has smaller percentage degradation, the memory bandwidth utilization is greatly reduced. Given that S-App often utilizes almost full peak memory bandwidth [61] and its performance largely depends on the memory performance in solo execution, a 20% degradation in `Worst` indicates that the system allocates 10.2GB/s (=20%× 4×12.8GB/s) memory bandwidth to the co-run NS-App. However, the MPKI (memory accesses per kilo instructions) of the NS-App in `Worst` is 24, i.e., 0.2GB/s bandwidth demand at most, which is much lower than the actually allocated amount.

To exploit the consolidation potential while mitigating the performance degradation, we analyze the co-run in details and identify the root causes of the degradation to each type of the applications.



(a) Path ORAM has synchronous progress across channels

(b) Requests from NS-App waste memory bandwidth

Figure 9: S-App has phase barriers and wastes memory bandwidth if out of synchronization.

3.4.1.1 Root Cause of S-App Degradation The S-App, to maximize channel utilization, maps the blocks in each bucket to different channels [61]. This leads to the synchronous progress across different channels, as shown in Figure 9(a). [61] showed that the memory utilization is close to peak bandwidth with synchronized progress. In Path ORAM, each ORAM access includes a read phase and a write phase. For security reasons, the write phase cannot start before the read phase completes. Also, a new ORAM access cannot start be-

fore the preceding one completes. Therefore, the end of each phase effectively becomes the synchronization barrier for all channels. This is referred to as **phase barrier** in this work.

In Figure 9(b), if one channel is slowed down due to scheduling the memory requests from NS-App, other channels need to wait even if they finish early. This leads to large memory bandwidth waste and significantly slows down S-App because the performance of the latter depends mainly on memory performance.

3.4.1.2 Root Cause of NS-App Degradation

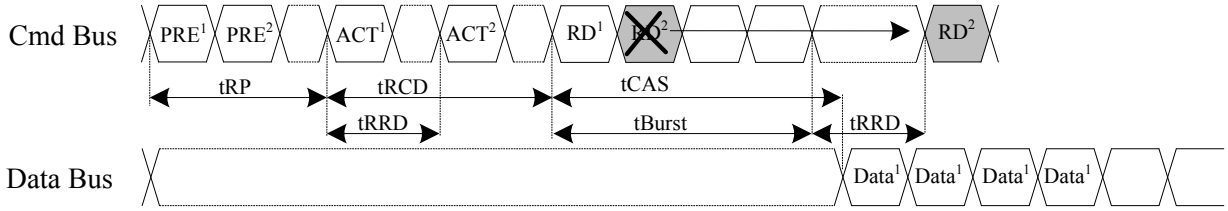
As discussed, the memory access intensity of NS-App is much lower than that of S-App. Given that the performance degradation of NS-App comes mainly from memory bandwidth competition, we next study the root cause by studying the memory scheduling details. Path ORAM adopts open page policy for better performance [61] — servicing a read request normally needs a `PRE` command to close the current row in the target bank, a `ACT` command to activate the row to be accessed, and a `RD` command to fetch the requested data. The baseline adopts FR-FCFS (first-ready first-come-first-serve) [53, 63, 62] with the goal to maximize memory throughput.

Table 1 lists a subset of timing constraints and their default values in the discussion.

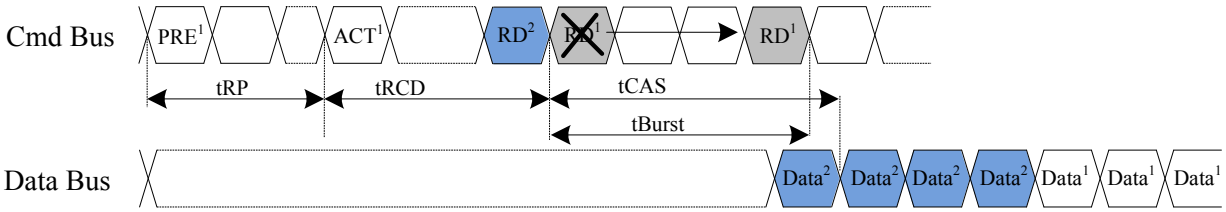
Table 1: Selective DRAM Internal Timing Constraint(at 800MHz)

Timing	Cycles	Description
tRP	11	Row Precharge. The time interval that it takes for a DRAM array to be precharged for another row access.
tRCD	11	Row to Column command Delay. The time interval between row access and data ready at sense amplifiers.
tRRD	5	Row activation to Row activation Delay. The minimum time interval between two row activation commands to the same DRAM device. Limits peak current profile.
tFAW	32	Four (row) bank Activation Window. A rolling time-frame in which a maximum of four bank activations can be engaged.
tCAS	11	Column Access Strobe latency.The time interval between column access command and the start of data return by the device.
tBurst	4	Data burst duration. The time period that data burst occupies on the data bus.

Figure 10 presents the memory bus timing for a co-run scenario with two applications. In the example, we assume two read requests `R1` and `R2` are from different types of applications, arrive at the memory controller at the same time, and demand data from different banks.



(a) 1 cycle delay of PRE^2 results in (tRRD+tBurst-1) cycle delay of RD^2



(b) A row buffer hit from S-App (i.e., RD^2) defeats NS-App priority

Figure 10: The co-run timing examples of two applications (CMD^i is the command for `Ri` (i=1 or 2)).

Since there is no constraint restricting precharge commands, the memory controller sends out PRE^1 and PRE^2 consecutively. PRE^1 is sent first because it has higher priority (which will be elaborated later in this section). However, due to timing constraints in JEDEC standard, ACT^1 needs to be tRP after PRE^1 while RD^1 needs to be tRCD after ACT^1. The interference enlarges due to additional constraints, e.g., ACT^2 has to be tRRD after ACT^1 and no more four `ACT` can be issued within tFAW time window; RD^2 needs to at least tBurst after RD^1 to avoid conflict on data bus. In summary, a later scheduled request tends to suffer from larger delay.

Since the baseline does not assign priority to either application, S-App and NS-App compete for the memory bandwidth based on their request arrival time. However, S-App translates each user memory access to tens of physical memory accesses, which gain more

opportunities to grab the system resources once the latter become idle. An NS-App request, when arriving at the memory controller, often gets delayed because the requested resources, e.g., the memory channel, are busy.

To mitigate the extremely biased request arrival in S-App and NS-App co-run, we have to assign high priority to NS-App to ensure that it gains sufficient opportunities to have its requests timely processed.

Scheduling NS-App with high priority. We reevaluate the experiment for Figure

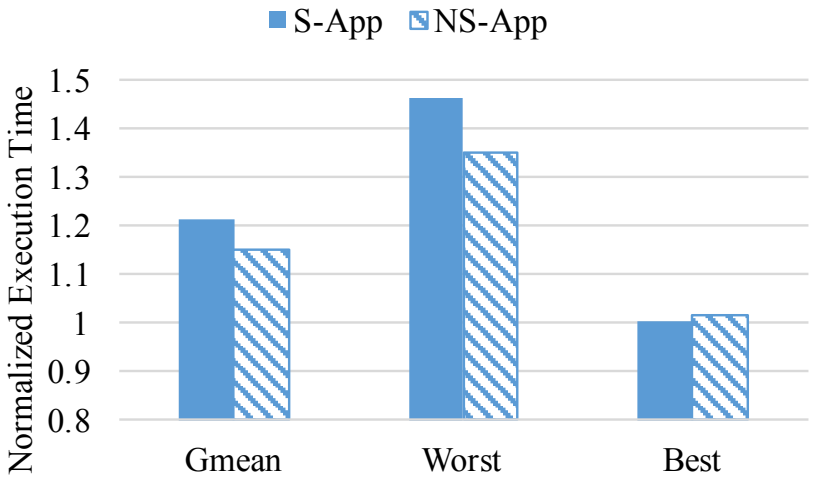


Figure 11: The co-run interference when NS-App has high scheduling priority.

8 with the scheduling priority assigned to NS-App. The results are summarized in Figure 11. From the figure, NS-App still suffers from 15% performance degradation on average. In addition, S-App shows much bigger degradation, rising to 21% on average and 46% in the worst case.

We analyze the memory scheduling details in the new experiment. We identify that it is the high row buffer hits in Path ORAM[61] that defeat simple priority allocation during scheduling.

Figure 10(b) illustrates the command sequence when servicing an NS-App request. Assuming `R1` is from NS-App and is a row buffer miss, the memory controller sends out `PRE`, `ACT`, and `RD` to service the request. In the example, a request `R2` (from S-App) arrives after `R1`. However, `R2`, when it is a row hit, which is frequent for S-App, can send out its `RD`[2]

command before RD[1], without PRE and ACT, and leads to extra delay to NS-App. In this case, NS-App still suffers from large performance degradation. From the discussions above, it is clear that while it has great potential to consolidate S-App and NS-App on one physical server, simple scheduling schemes, either with or without priority, tend to introduce large performance degradation and memory bandwidth waste.

3.5 THE CP-ORAM DESIGN DETAILS

3.5.1 P-Path: enforce scheduling priority through resource pre-allocation

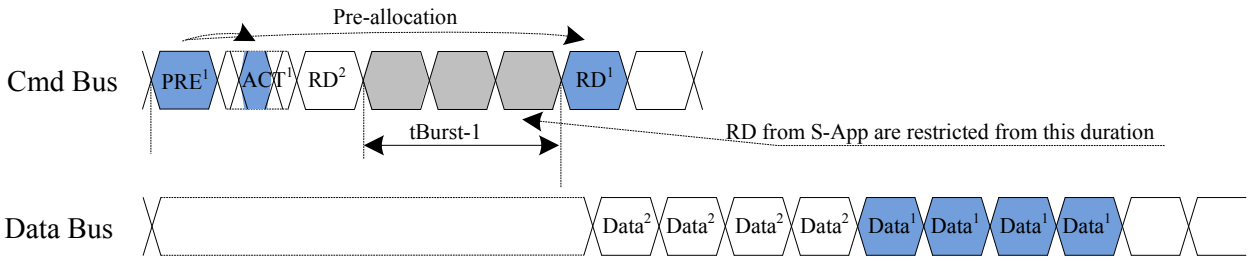


Figure 12: P-Path pre-allocates bus slots to enforce priority allocation for NS-App.

To address the large performance loss in the co-run, we propose CP-ORAM that consists of three cooperative memory scheduling schemes — P-Path, R-Path, and W-Path. We elaborate P-Path in this section and R-Path and W-Path in following sections.

P-Path is designed to assign scheduling priority to NS-App and enforce this priority through resource pre-allocation. The simple allocation in Section 3.4.1, while prioritizing PRE, cannot prevent ACT and RD commands from being delayed by S-App requests. The P-Path scheme addresses this issue by resource pre-allocation. That is, when the memory controller sends out PRE for an NS-App read request, it also allocates address, data, and command bus slots for its coming ACT and RD commands. S-App requests, even if they are row buffer hits, cannot be scheduled if otherwise leading to resource conflict. Figure 12 illustrates that an S-App RD for a row buffer hit cannot be scheduled within (tBurst-1) cycles from the pre-allocated RD slot.

While pre-allocation helps to enforce priority allocation for NS-App, it may degrade S-App performance significantly. Therefore, as shown in Algorithm 1, P-Path proportionally pre-allocates channel and bus resources based on a given threshold `th` (percentage value) and pre-allocates for every `th`×`10` out of 10 row buffer miss requests from NS-App.

The threshold `th` is statically determined in this work. We leave it as our future work to develop dynamic threshold adjustment for better trade-off among resource utilization and performance improvement. The algorithm gives NS-App higher scheduling priority, i.e., if an S-App request/device command and an NS-App request/device command are ready at the same time, the priority is always given to NS-App. The algorithm is enabled only for row buffer misses from NS-App, the requests with row buffer hits do not need resource pre-allocation.

3.5.2 R-Path: maximize memory bandwidth utilization using next read

From the analysis in Section 3.4.1, the performance degradation of S-App comes mainly from the phase barriers. That is, the progress of multiple memory channels may lose synchronization due to co-run interference; if the slowest channel has not reached its phase barrier, other faster channels have to wait, leading to significant waste of memory bandwidth. In this section, we propose R-Path to maximize memory bandwidth utilization using the read operations from next ORAM access.

R-Path is designed to schedule memory operations across phase barrier to improve bandwidth utilization. Figure 13 elaborates what operations may be promoted without tampering with the correctness as well as the security of Path ORAM. In the example, we assume two consecutive Path ORAM accesses need to access blocks `b1` and `b2` on paths `l1` and `l2`, respectively. Along path `l1`, there might exist a block `b3` from path `l3`. Paths `l1` and `l3` have significant overlap while paths `l1` and `l2` only overlap at a level close to the root. For simplicity, we assume `b1` and `b2` are mapped to paths `lx` and `ly`, respectively, after the accesses and these two paths have no overlap with `l1` and `l3` other than the root bucket (so that they are ignored from discussion). Since Path ORAM tries to push blocks as deep as possible in the write phase, `b3` may replace `b1`'s place. Given that we do not know if `b3`

Algorithm 1: P-Path Scheduling Algorithm

Input: PreAllocation Threshold `th` (percentage value)

Output: Issue proper command to DRAM

Parameter: `cycle`: program cycle;

`cnt`: row buffer misses from NS-App

```
1  while not end of program do
2      if can issue memory commands at cycle then
3          check channel command queue;
4          if has commands from NS-App then
5              issue the command;
6              if the command is PRE, i.e., being a row buffer miss then
7                  cnt = cnt++ % 10 ;
8                  if cnt < th×10 then
9                      pre-allocate address, command, and data bus slots for ACT and RD/WR;
10                 end
11             end
12         else
13             issue S-App command;
14         end
15     else
16         continue;
17     end
18     cycle++;
19 end
```

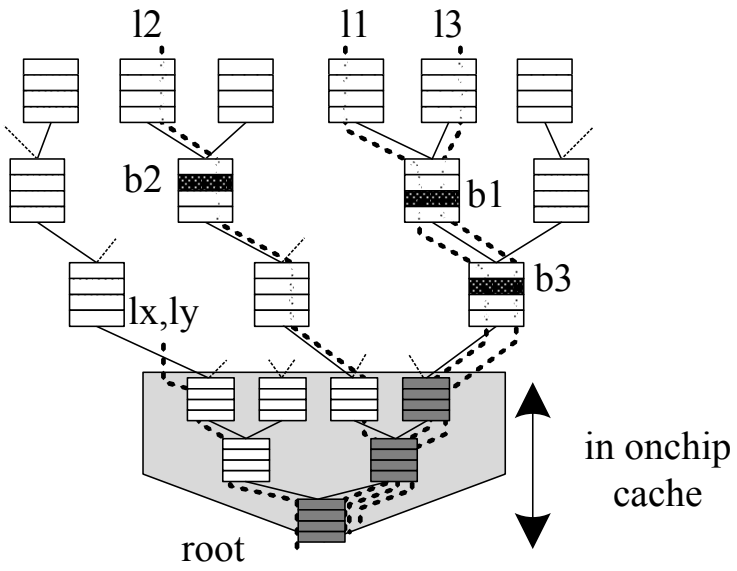


Figure 13: R-Path safely promotes reads from the next ORAM access.

exists until we read all blocks from `l1`, the write phase is tightly data dependent on the read phase. Therefore, we cannot schedule operations from the write phase to the read phase in the same ORAM access.

Interestingly, the blocks read during the read phase of the second ORAM access, i.e., reading path `l2`, do not have data dependence. In this example, path `l1` and `l2` overlap at a level that is close to the root bucket. Since the top 10 levels of the tree are cached, accessing the overlapped buckets does not lead to memory accesses. Thus, it is safe to schedule read operations from the second ORAM access to improve the bandwidth utilization. This effectively prefetches data blocks from the read phase of the next ORAM access.

To ensure the correctness of Path ORAM, R-Path works as follows.

- R-Path only prefetches data blocks from the read phase of its immediate next ORAM access and stores the prefetched blocks in a small read buffer. The prefetched blocks cannot be sent to stash directly as, otherwise, some blocks may be selected for write back to path `l1` in write phase. The latter is infeasible when there is no prefetch.

  In this work, each ORAM path stores 14 (=24-10) levels of the tree in memory. Therefore,

the prefetch buffer needs to hold at most 56 blocks. The prefetched data are copied to the stash after Path ORAM has determined what to write for the write phase of the current access.

- R-Path creates a dummy access if there is no ORAM access in the queue, similar to that in [50, 21].

  If the 2nd ORAM access overlaps with the 1st one in memory, i.e., at a level ≥ 10, we enable the fork path optimization [95] and disable R-Path. Our experimental results show that the probability of the former is very low due to its low probability after large tree top caching.

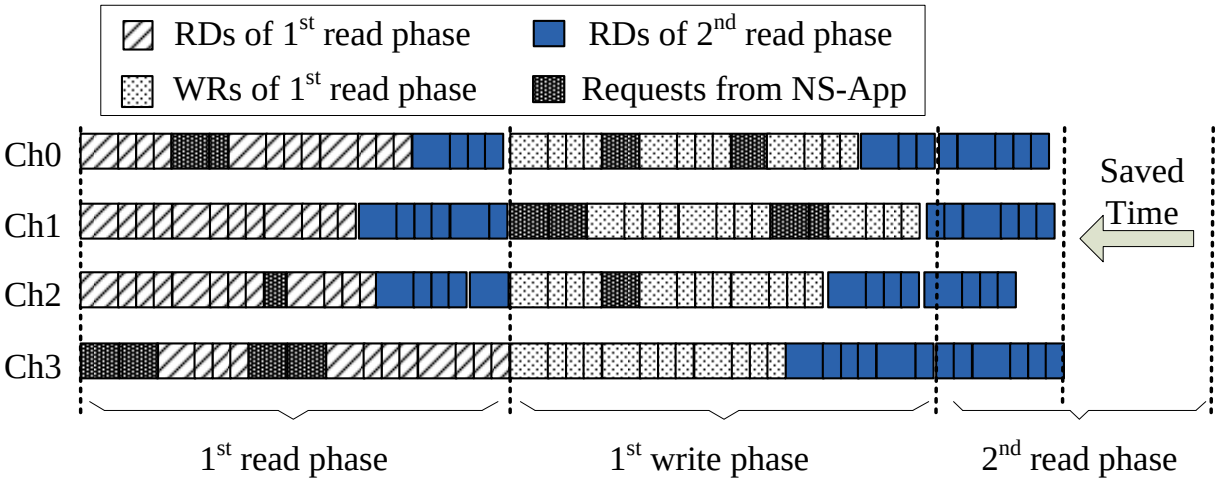


Figure 14: R-Path improves memory bandwidth utilization and speeds up program execution.

As shown in Figure 14, R-Path promotes read operations from the 2nd ORAM read to the 1st read or write phases, which improves the memory bandwidth utilization of the 1st read and write phases, and shortens the read length of the 2nd ORAM access.

3.5.3 W-Path: mitigating write traffic on busy channels

R-Path shortens the length of read phase by promoting some of its operations to preceding phases. It cannot reduce the length of write phase because write operations cannot be

promoted across the read/write phase barrier in one ORAM access. In this section, we elaborate the W-Path design to mitigate the co-run interference in the write phase of Path ORAM access.

Intuitively, W-Path schedules write operations across phase barriers by deferring them to future write phases. In this way, write operations can also be moved across the phase barriers, but not the read/write phase barrier in the same access. Figure 15 illustrates how it works.

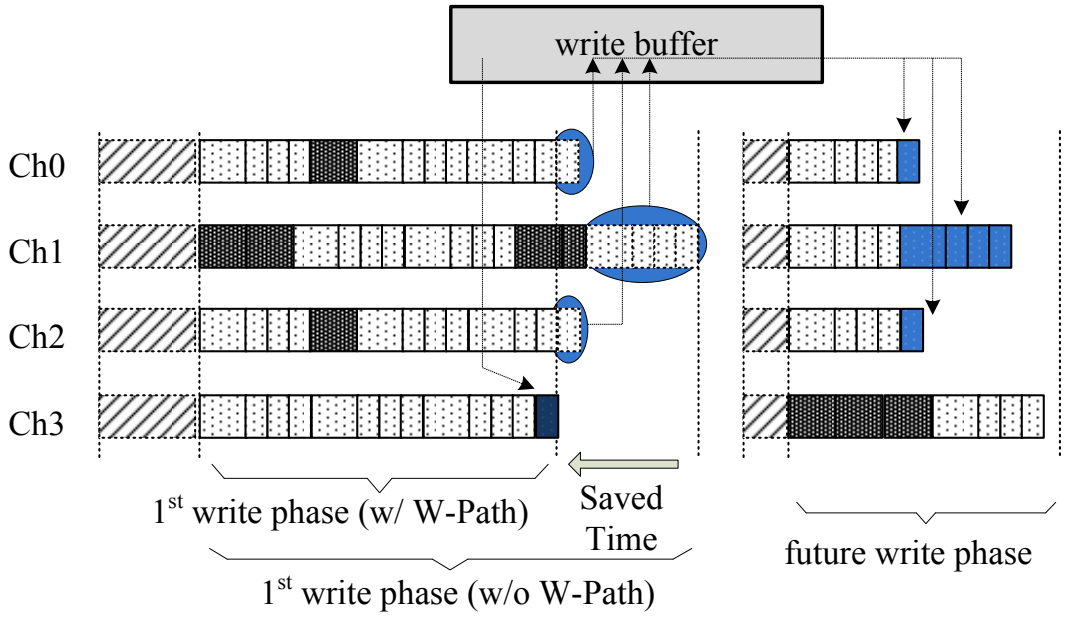


Figure 15: W-Path uses a write buffer to schedule write operations across phase barriers.

- W-Path is triggered when one or multiple memory channels finish their write operations for the current ORAM access. W-Path searches for the data blocks in the write buffer that need to be written to the fast channels and starts to expunge them. Their slots in the write buffer are then identified as empty entries.
- Given that one ORAM access writes back 14 blocks to each channel. We integrate a 56-block write buffer. The write buffer reserves 8 entries for each channel and let all channels compete for the rest of the entries.
- W-Path aggressively uses the write buffer to shorten the length of the write phase. If a channel has data blocks in its ready queue, and the write buffer has empty reserved

entries for this channel, W-Path moves one block from the corresponding channel queue to the write buffer. If the reserved entries in the write buffer for the channel are full, W-Path moves blocks only if doing so helps to reduce the length of the write phase.

- The write buffer is looked up during the read phase such that a hit in the write buffer gets the data from the write buffer directly.

As a comparison to R-Path that can move read operations across phase barriers to the read and write phases of the preceding ORAM access, W-Path temporarily buffers the write operations in the write buffer such that they are defer to future ORAM accesses.

3.5.4 Architectural enhancements

Figure 16 presents an overview of CP-ORAM design. The shaded boxes indicates the components that are either added or enhanced. We enhance the DRAM controller to enable fine-grained priority enforcement. The read buffer (for R-Path) and write buffer (for W-Path) are added into the ORAM controller. In this work, each buffer can store up to 56 blocks, i.e., 4KB each. In the read phase, for the current ORAM access, the write buffer is looked up with matched blocks sent to stash. At the end of write phase, i.e., the beginning of the next read phase, the blocks in read buffer are sent to stash. Both operations are in parallel with the accessing of remaining blocks (of the current ORAM access) from the memory. In the write phase, the write buffer is visited when a block need to be swapped out. Since the write buffer is very small (14 entries per channel), finding a block to be written out is relatively fast, and imposes negligible accessing overhead.

The secure processor can switch between secure mode and non-secure mode. The former demands encryption and Path ORAM while the latter has no security enforcement. We assume there is no address space overlap between S-App and NS-App, and one bit is communicated to the memory controller to differentiate S-App memory accesses from NS-App ones.

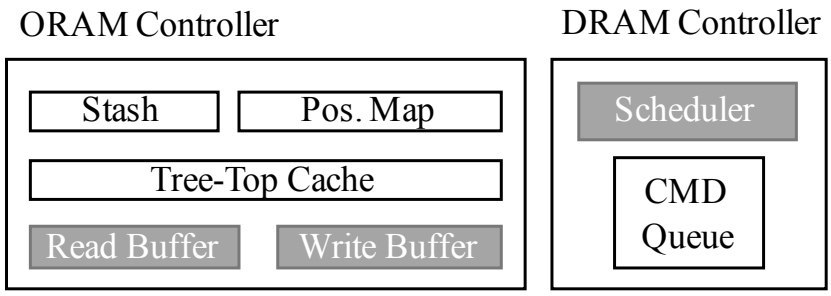


Figure 16: Architectural enhancements for CP-ORAM.

3.5.5 Security analysis

ORAM prevents information leakage with address randomization – it assigns a new physical address to each program address accessed. Our cooperative designs do not modify this strategy and thus do not compromise the security guarantee of Path ORAM.

The first scheme, P-Path, modifies the scheduling priority based on a simple counter. The resource pre-allocation policy only affects the completion time but dose not reveal additional pattern information. The second scheme, R-Path, prefetches data blocks from the following ORAM access. R-Path is enabled only if the paths of two accesses do not overlap at a level in memory. Therefore, the data blocks from the 2nd path need to be read anyway. The exception is that the second ORAM access becomes ready after the read phase the preceding one — in this case, a dummy access shall be inserted, which delays the second ORAM access by one ORAM access length. As discussed in [95], this does not degrade security. Due to large top cache we use, we did not observe the merging opportunity as in [95]. The W-Path does not compromise security guarantee either. W-Path alters the timing of the write operations, but not the contents. The encrypted data blocks are eventually written back to the memory, the same as the baseline.

In addition, our proposed schemes do not increase stash overflow probability. The read buffer and write buffer ensure the stash remains the same as the baseline.

3.6 EXPERIMENTAL METHODOLOGY

Table 2: Baseline System Configuration

Parameter	Value
Processor	Dual-core, 3.2GHz
Processor ROB size	128
Processor retire width	4
Processor fetch width	4
Last Level Cache	512 KB per core
Memory bus speed	800MHz
Memory channels	4
Ranks per channel	1
Banks per rank	8
Rows per bank	16384
Columns (cache lines)/row	128
Memory Space	4GB + 4GB

To evaluate the effectiveness of CP-ORAM, we used USIMM, a cycle accurate memory system simulator [8], to simulate the proposed schemes and compare them to the state-of-the-art. Table 2 summarizes the baseline server configuration on which we co-ran two applications — one S-App and one NS-App. The DRAM memory follows JEDEC DDR3-1600 specification. We adopted the default values in the specification that are strictly enforced in USIMM.

In the baseline setting, each application has its own 4GB memory space. The address mapping follows the order of "`row:bank:column:rank:channel:offset`" such that Path ORAM maximizes its row buffer hit and both applications fully utilize all channels, as shown in [61].

We chose a set of memory intensive benchmarks from PARSEC suite, commercial, SPEC and BioBench, as they were used in MSC [40]. Each benchmark is simulated for 5 billion

instructions, and 500 million representative instructions were selected with a methodology similar to Simpoint [40]. We constructed the workloads for evaluation as follows: each workload consists of one S-App and one NS-App that are of the same program: S-App version adopts encryption and Path ORAM protection while NS-App version does not. Their visible physical address sequences are completely different. We also show other combinations in the detailed result analysis. Table 3 describes the benchmark programs. The MPKI (memory access per kilo instructions) is listed in parenthesis. We use the first two letters of each program to indicate the workload.

Table 3: Simulated Benchmark Programs

Suite	Workloads
PARSEC	black (4.2), face (26.8), ferret (8.0), fluid (17.5), freq (4.5), stream (12.9), swapt (10.9)
COMM.	comm1 (7.3), comm2 (12.6), comm3 (4.2), comm4 (3.7), comm5 (4.5)
SPEC	leslie (23.1), libq (12.0)
BIOBENCH	mummer (24.0), tigr (6.7)

3.7 RESULTS

In the experiments, we evaluated the following schemes:

- `Baseline`. This is the baseline for comparison purpose. We collected S-App and NS-App performance, respectively, in their solo execution mode.
- `FR-FCFS`. This is the co-run case that adopts FR-FCFS memory scheduling algorithm. Each workload consists of S-App version and NS-App version of the same program.

- `P-Path`. This is the co-run case that adopts P-Path scheduling enhancement. The threshold `th` used in P-Path is set to 50% by default.
- `P+R-Path`. This the co-run case that adopts P-Path and R-Path scheduling enhancements.
- `P+R+W-Path`. It adopts all three cooperative Path-ORAM scheduling enhancement, i.e., P-Path, R-Path and W-Path.

3.7.1 Performance analysis

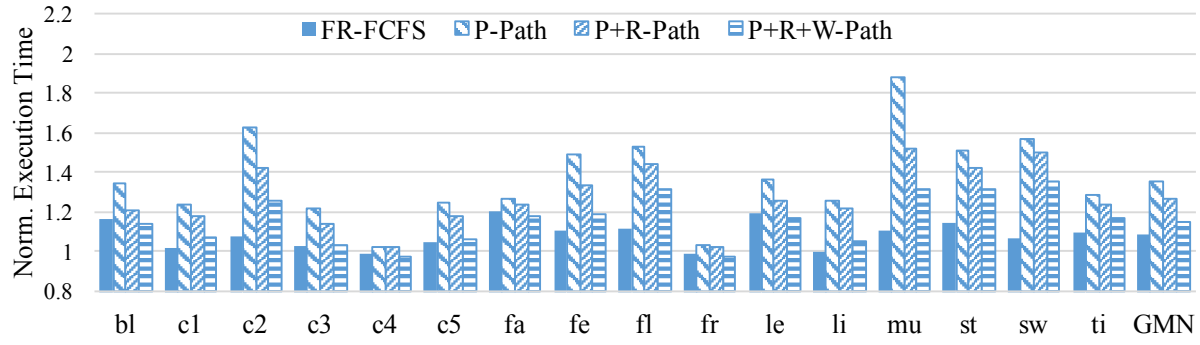


Figure 17: The normalized execution time for S-App.

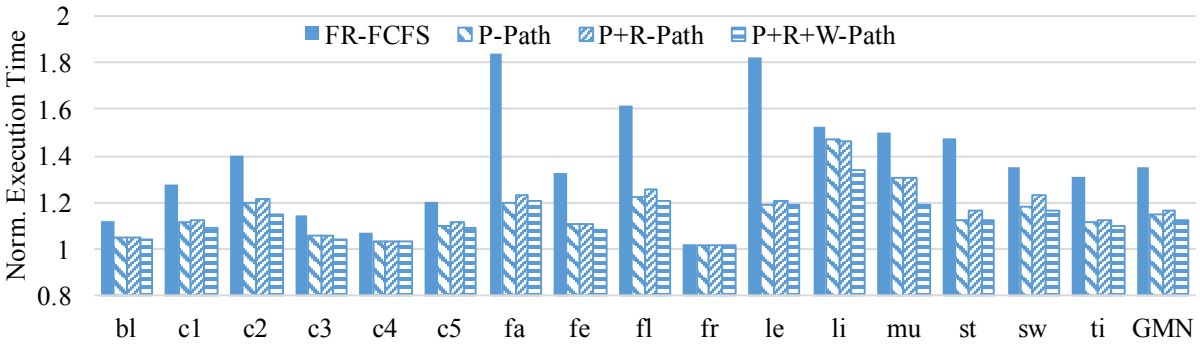


Figure 18: The normalized execution time for NS-App.

We first compared the effectiveness of different CP-ORAM schemes with the results summarized in Figure 17 and Figure 18 for S-App and NS-App, respectively. The results

are normalized to the solo execution `Baseline`. From the figure, we found that FR-FCFS leads to large performance degradation to both S-App and NS-App — on average, 8.3% and 35.5%, as shown in the motivation section. S-App and NS-App may suffer up to 20% and 84% degradation, respectively.

`P-Path`, while reducing the degradation from 35.5% to 15% for NS-App, significantly increases the degradation for S-App, i.e., from 8.3% to 35.2% on average. Our R-Path and W-Path schemes target at improving S-App performance. For S-App, `P+R-Path` reduces the performance loss to 26.3% on average while `P+R+W-Path` further reduces it to 15.4% on average. That is, CP-ORAM achieves around 20% performance improvement over the FR-FCFS scheduling for S-App.

In general, the R-Path and W-Path schemes have little impact on NS-App. When R-Path promotes read operations from the next ORAM access, the memory bandwidth utilization is improved, which slightly hurts NS-App. We observed around 1% extra degradation. When W-Path defers write operations to future ORAM accesses and thus move to write buffers, the channel is less busier, which helps NS-App. We observed an average of 2.4% reduction. In summary, CP-ORAM reduces the the performance degradation from 15% to 12.6% for NS-App on average.

3.7.2 P-Path pre-allocation threshold

By default, `P-Path` pre-allocates resources for half of NS-App row buffer misses, i.e., `th`=50%. We studied the performance impact with different threshold values and summarized the results in Figure 19. The x-axis is the pre-allocation threshold. In general, larger threshold values results in larger performance degradation to S-App and smaller degradation to NS-App.

Also from the figure, R-Path and W-Path improve S-App performance significantly, but has little impact on NS-App. This is because improved bandwidth utilization benefits S-App the most. The maximal gain (sum of percentage improvements from both S-App and NS-App) occurs at `th`=80%, in which the workload achieves 21.6% sum in total.

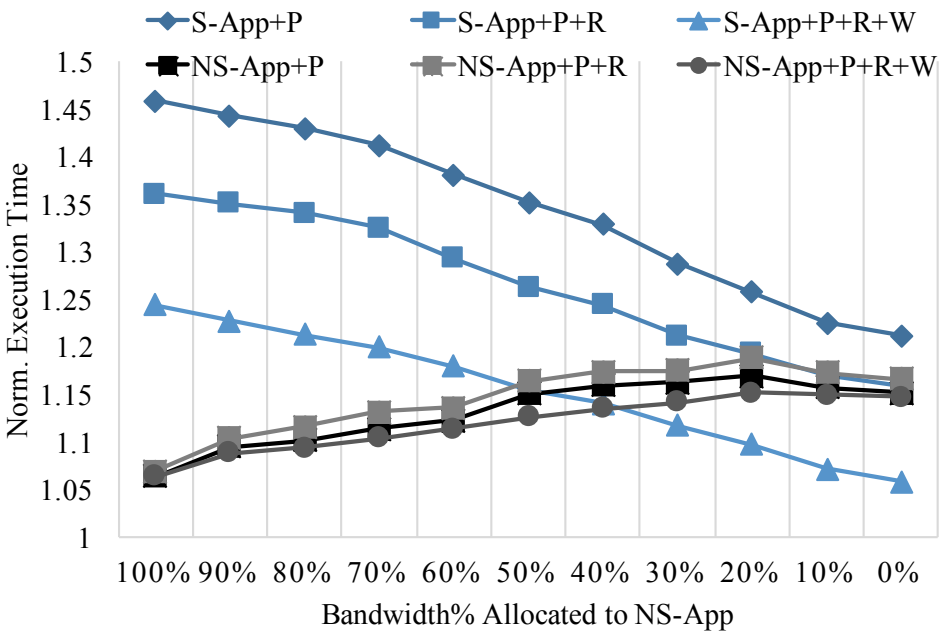


Figure 19: The effectiveness of CP-ORAM with different threshold values.

3.7.3 Memory access latency

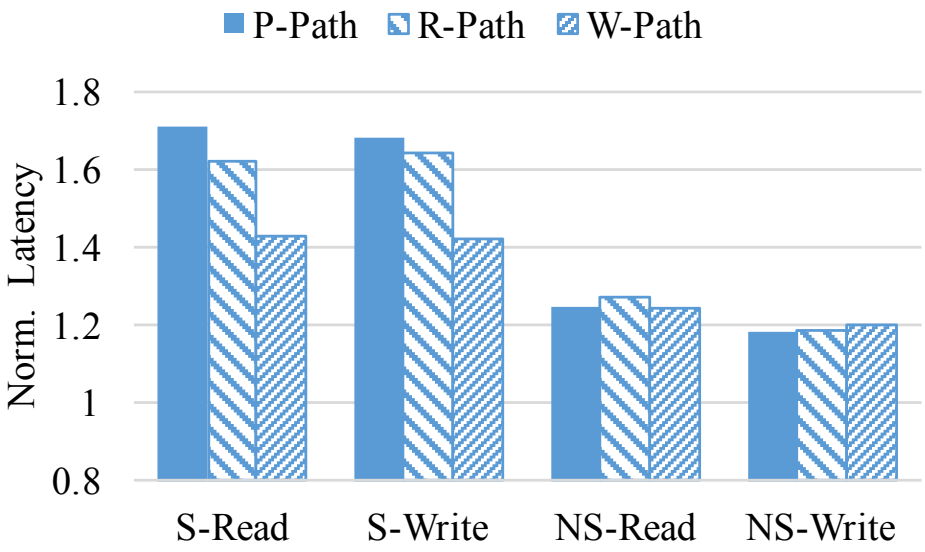


Figure 20: Comparing memory access latency under different schemes.

Figure 20 reports the average read and write latencies for S-App and NS-App under different schemes. All results are normalized to the solo-run. From the figure, R-Path shows larger reduction on read latency than that on write latency. This is because R-Path prefetches read operations and thus shortens the read phase in general. For S-App, W-Path shows more reduction on both read and write latencies over R-Path. This is because W-Path shortens the current write phase such that pending requests are serviced early.

For NS-App, both R-App and W-App have little impact — the difference across three schemes is less than 3% for read latency and 1% for write latency.

3.7.4 Buffer usage

We then analyzed the effectiveness of the read buffer and the write buffer. Figure 21 reports the average number of blocks prefetched per channel. From the figure, R-Path prefetches more than 4 blocks per channel on average, with the maximum being around 9 for `leslie`. `comm4` and `ferret` prefetch around 1 block per channel per access, leading to negligible

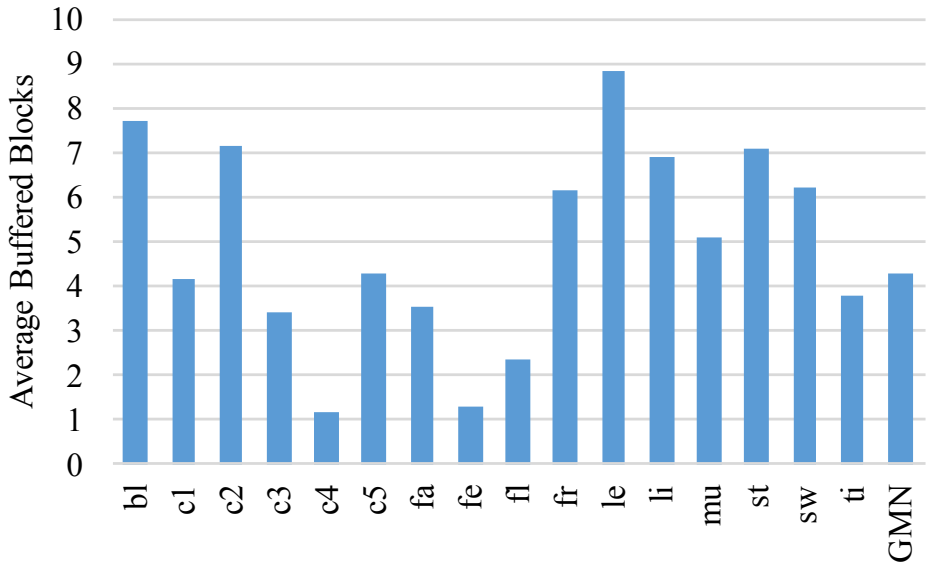


Figure 21: The average number of blocks per channel in read buffer.

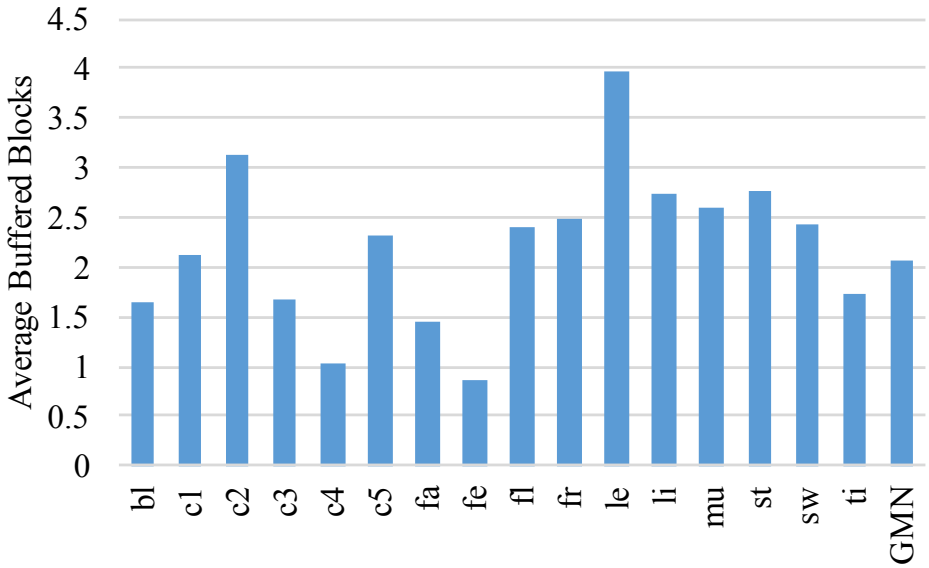


Figure 22: The average number of blocks per channel in write buffer.

performance improvement in Figure 17. We found that the more imbalanced channel use the NS-App brings to the system, the more opportunities R-Path has to prefetch next read operations.

For W-Path, Figure 22 reports the average number of blocks in write buffer per channel. We observed the similar trend — with more blocks deferred, W-Path gains better improvements.

3.7.5 Sensitivity to core number

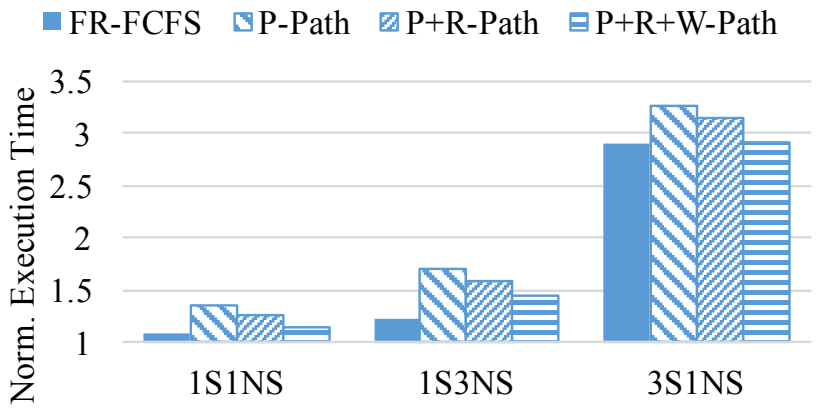


Figure 23: Comparing S-App performance with more co-running applications.

At last, we studied the effectiveness of CP-ORAM design with more than two cores. We compared two settings using four cores: one co-runs one S-App and three NS-App applications; the other co-runs three S-App and one NS-App applications, referred to as 1S3NS and 3S1NS, respectively. Figure 23 and 24 report the geometric mean of normalized performance results from 16 workloads. P-Path adopts `th`=50%. 1S1NS refer to default setting that was used in previous experiments.

From the figure, we observed larger performance degradation when there are more applications. Given that one S-App uses almost all memory bandwidth, 3S1NS has extreme intensity and introduces close to 3× performance degradation to S-App. On the other hand, 1S3NS has more bandwidth demand than 1S1NS but much less than 3S1NS — we observed slightly larger degradation than 1S1NS but much less than 3S1NS.

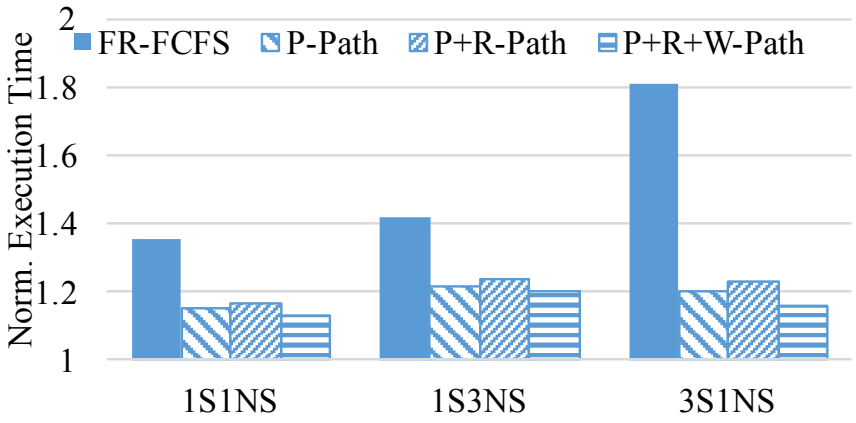


Figure 24: Comparing NS-App performance with more co-running applications.

In all cases, our cooperative scheduling schemes are robust. `P-Path` reduces the NS-App degradation to around 20% degradation while `P+R+W-Path` reduces more than 20% slowdown over `P-Path` for S-App.

3.7.6 More co-run examples

In this section, we present more co-run examples results. The first study is to co-run different S-App and NS-App. Figure 25 and 26 shows the normalized execution of this co-run scenario. For x-axis label, the first application is secure and the second one is non-secure. We still set the threshold as 50% in this case. We found similar performance compared to the main results in Section3.7.1. Overall, `P+R+W-Path` reduce S-App 19.8% execution time compared to `R-Path` only, and is only a slight increase of 7.1% over FR-FCFS. The overall technique also reduce NS-App execution time of 22.9% compared to FR-FCFS.

We also studied co-run effect of multiple copies of application on the same memory channel configuration. Figure 27 shows the normalized execution time across all workloads when there are multiple copies of S-App and NS-App running. It can be observed that co-run 2 and 4 S-App will cause 1.78x and 3.46x more execution time, while co-run 2 and 4 NS-App will only cause 5.6% and 13.8% more time.

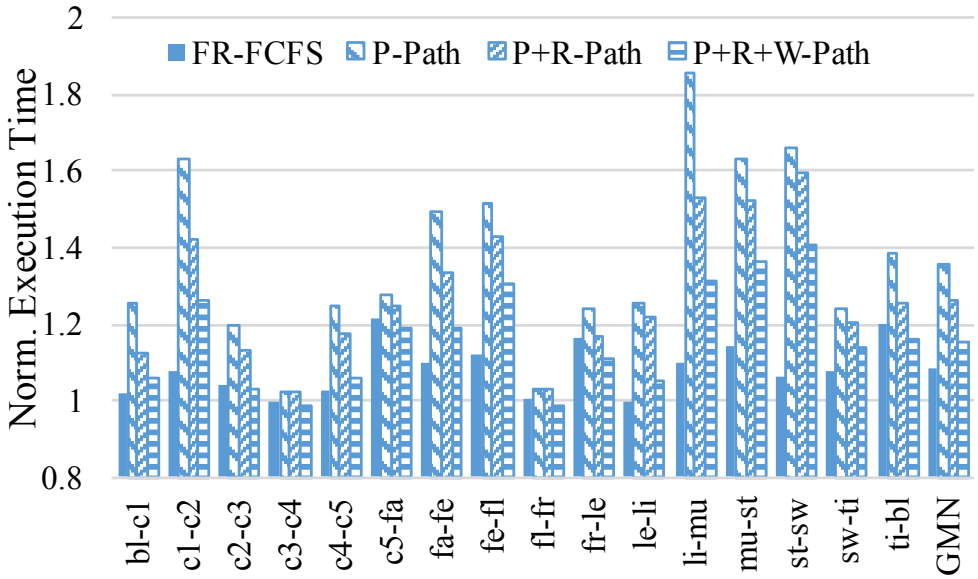


Figure 25: Comparing S-App performance with mixed co-running applications.

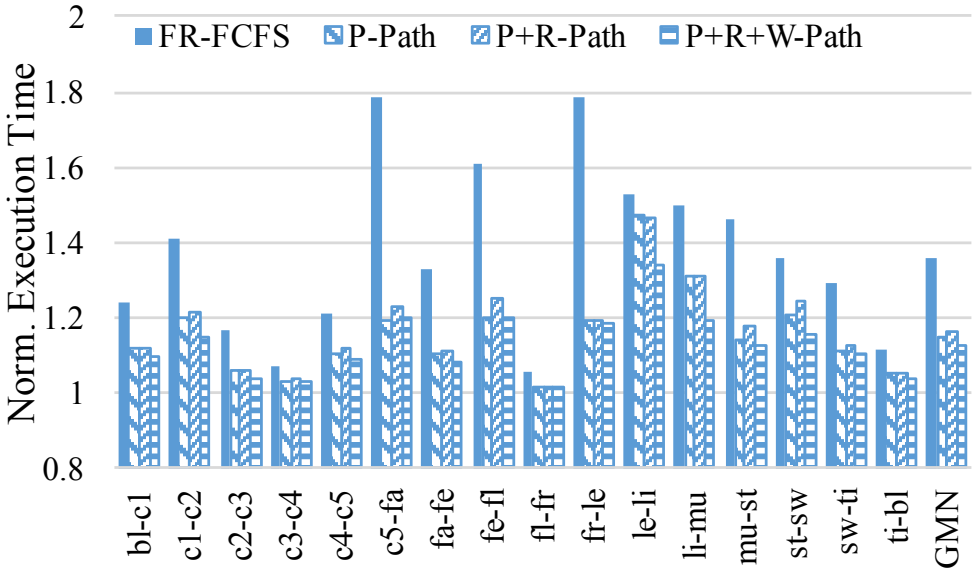


Figure 26: Comparing NS-App performance with mixed co-running applications.

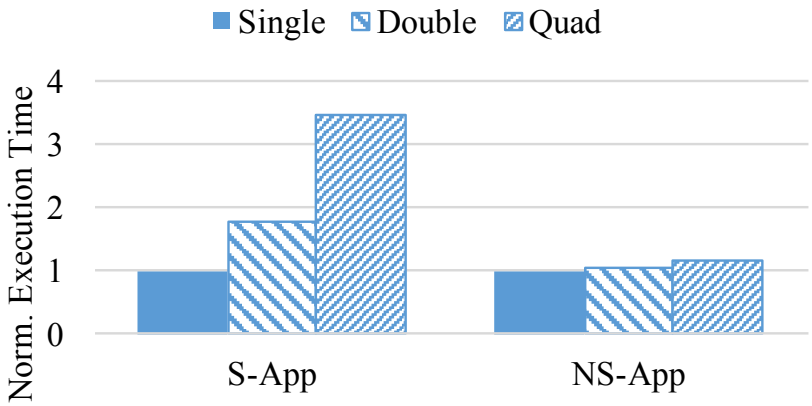


Figure 27: The normalized execution time for NS-App.

3.8 CONCLUSIONS

In this chapter, we propose CP-ORAM, a Cooperative Path ORAM design, to address the above issues. We summarize our contributions as follows.

- We study the co-run interference between secure and non-secure applications and analyze the root causes of the ineffectiveness in adopting traditional memory scheduling. To our knowledge, this is the first work that focuses on memory scheduling for Path ORAM in server settings.
- We propose CP-ORAM that consists of three cooperative scheduling schemes for effective memory bandwidth sharing. *P-Path* is designed to assign and enforce scheduling priority during the co-run. *R-Path* maximizes channel utilization by proactively scheduling read operations from the following Path ORAM access. *W-Path* mitigates contention on busy channels by writes redirection.
- We evaluate CP-ORAM and compare it to the state-of-the-art. Our experimental results show that CP-ORAM achieves an average of 20% performance improvement over the baseline Path ORAM for the secure application in a server setting with four channels.

4.0 REMOVING INTERFERENCE WITH ORAM DELEGATOR

4.1 THREAT MODEL

To facilitate security analysis, the system components of a cloud server are often partitioned to those that are trustworthy, i.e., the trusted computing base (TCB), and those that are not [35]. A system is *secure* if all attacks from outside of TCB can be successfully defended. As an example, if the OS is in the TCB, there is no need to defend attacks from the OS kernel. However, including potentially an untrustworthy component in TCB could break the security guarantee and leave the system in a vulnerable state. A curious or malicious (after being hijacked) OS can easily break the security mechanisms that the application may adopt.

Executing a secure program in an untrusted environment such as on the cloud server faces various types of attacks. Therefore, the TCB is preferably as small as possible. Following the threat model in previous studies [61, 23, 95], the OS is not in the TCB and physical attack is possible. A curious OS may launch profiling code to collect execution statistics; a malicious OS may record keystrokes or sensitive data used during the execution. In particular, an attacker may attach physical devices to analyze the communication signals.

In this work, the processor chip is included in the TCB, similar to previous designs [44, 61, 1, 4]. In addition, a hardware component X can be optionally placed in the TCB such that TCB consists of CPU and X. We next compare different designs to illustrate their tradeoffs.

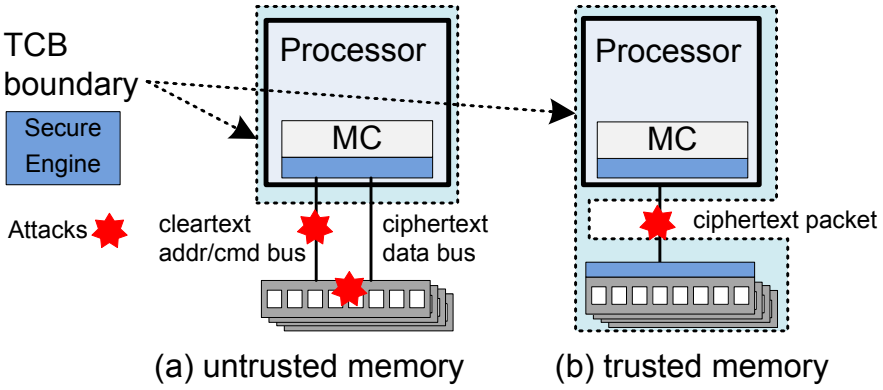


Figure 28: Comparing two TCB models.

4.1.1 The model assuming trusted processor and untrusted memory

When TCB includes the processor chip only, i.e., no additional X component is in TCB, as shown in Figure 28(a), secure application execution needs to defend all attacks from outside of the processor chip such that data confidentiality, integrity, and privacy are protected during execution.

Adopting data encryption helps to enforce data confidentiality. Lie *et al.* proposed to encrypt the user code and data when they are saved in memory or disk and decrypted when being brought to the processor chip for computation. A secure engine is integrated into the processor chip to facilitate the cryptographic operations [44]. Suh *et al.* proposed Merkle tree based verification to efficiently check the integrity of memory that contains dynamic data [73].

However, it is challenging to prevent information leakage from memory accesses. To access data saved in the untrusted memory, the on-chip memory controller needs to convert a read or write request to a sequence of device commands. Since the memory module is not trustworthy, those commands, as well as the memory addresses, are sent in cleartext. Even though the data exchanged between the processor and the memory module are encrypted, the access pattern of memory addresses may leak sensitive information, e.g., when a medical application searches for the treatment information for a specific disease from the database,

it is likely that the current patient has corresponding symptoms [11]. Even when both code and data are unknown to the adversary, previous work has demonstrated a control flow graph (CFG) fingerprinting technique to identify known pieces of code solely based on the address trace [96].

ORAM Model. Studies have shown that to securely prevent information leakage from memory access patterns, it demands oblivious memory (ORAM) primitives [27, 28]. ORAM conceals the access pattern from an application by continuously shuffling and re-encrypting the memory data after each access. An adversary, while still being able to observe all the memory addresses transmitted on the bus, has a negligible probability to extract the real access pattern.

Path ORAM [72] was recently proposed as a practical ORAM implementation. Figure 4 shows the logic component and organization of a path ORAM protected system. The physical memory is organized as a binary tree with each node consists of, e.g., four, memory blocks (i.e., cachelines). The logic addresses are randomly mapped to tree paths with the mapping recorded in the *position map*. When there is an LLC (last level cache) miss, the position map is consulted to get the path number. Path ORAM fetches all physical blocks along the path. After reading and decrypting these blocks, the requested block can be returned to the LLC. It is then remapped to a different path and temporarily buffered in the *stash*. Other blocks of the path, together with a subset of blocks from the stash that can be merged to the path, are encrypted and written back to the memory. When caching the top of the tree in a small cache, the number of accesses can be reduced [61].

In summary, a Path ORAM access consists of read and write phases with each phase read and write all blocks of a tree path, respectively. Given 4GB Path ORAM tree, if each bucket contains 4 blocks, the tree has 24 levels such that one phase accesses 23×4 blocks if only the root node is cached, or 21×4 blocks if top 3 levels are cached, etc. These accessed blocks can be physically mapped to multiple memory channels to increase parallelism.

4.1.2 The model assuming trusted processor and trusted memory

An alternative TCB model is to place both the processor and the main memory module in the TCB, as shown in Figure 28(b). The recent proposed ObfusMem [4] and InvisiMem [1] schemes use this model.

Since the communication channel is not included in the TCB, the data exchanged between the trusted processor and the trusted memory still need to be encrypted and authenticated. A secure engine is integrated into the memory module to support cryptographic operations.

There is no need to adopt Path ORAM protection if the memory is trustworthy. A secure memory scheme encrypts the packets for protecting data confidentiality and generates the packets with the same length and order for both read and write request types. When there are multiple channels, the scheme needs to generate dummy requests to the channels other than the one that the data located.

The secure memory model works well with HMC architecture but faces challenges when applying to untrusted memory settings. As an example, adding a secure (bridge) chip to DRAM DIMM cannot meet the secure memory model requirement as the wires on the PCB may be compromised such that the communication between the secure chip and the memory chips is eavesdropped.

4.1.3 Comparing secure execution models

We next compare the two secure execution models and study the performance impacts in different settings. In the following discussion, we use the following abbreviation.

S-App — a trusted process that adopts either Path ORAM, secure memory model, or our model for protection; and

NS-App — a process that does not need protection.

Figure 29 summarizes the average execution time of NS-Apps when co-running one S-App and seven NS-Apps on an 8-core CMP when using different memory settings. For a suite of programs that we tested, we report the best, the worst, and the geometric mean cases. For the multiple process scenarios, we simulated multiple instances of the same workload in multi-programming fashion, similar to those in [40]. The architecture details are listed in

Section 4. `1NS` indicates the solo execution, i.e., there is no other co-run applications. `1S7NS` indicates that the eight processes compete for four memory channels. `7NS-4ch` and `7NS-3ch` indicate the channel partition that the seven NS-Apps compete for four and three memory channels, respectively, while the S-App uses a separate channel (with results not shown).

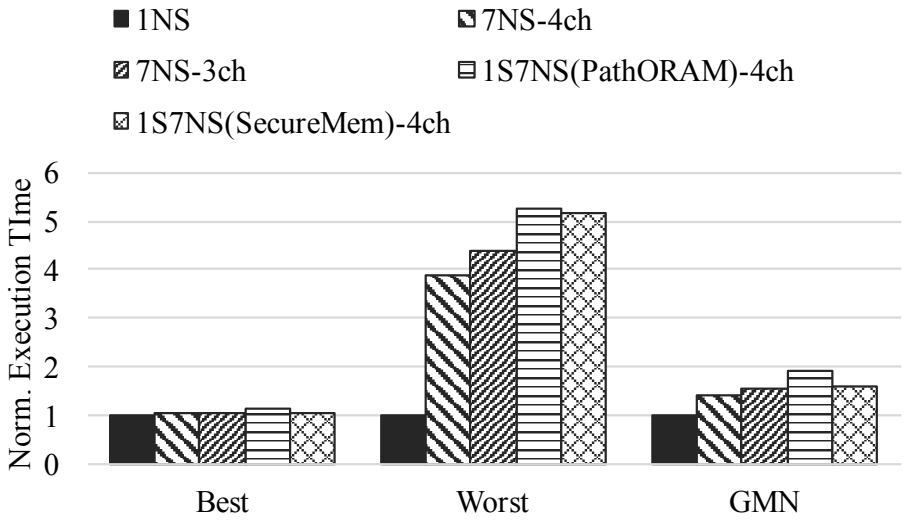


Figure 29: The performance degradation under different co-run scenarios.

From the figure, NS-Apps suffer from large performance degradation when the system has a co-run S-App. When we adopt Path ORAM, i.e., `1S7NS (Path ORAM)`, the non-secure application may take up to 5.26× execution time of the solo execution, and an average of 90.6% execution time overhead. Given that each application has individual core and cache resources, the interference comes mainly from the contention for the memory bandwidth. As shown in [61, 80], an S-App may consume close to 100% of the peak memory bandwidth, which introduces large performance degradation to co-running non-secure applications.

A potential optimization is to enable the channel partition, i.e., to allocate the Path ORAM accesses to one channel and allocate the seven NS-Apps to other three channels. While `7NS-3ch` achieves significant improvements compare to `1S7NS(Path ORAM)`, the degradation is still significant. On average, NS-Apps exhibit 57% slowdown compare with solo-run(1NS) performance. By giving one more memory channel bandwidth resource, `7NS-4ch` shows 43% slowdown, which is still significant.

On the other side, adopting secure memory execution model is beneficial but not significant. We modeled the channel replication and read write obfuscation as in ObfusMem[4] and InvisiMem[1]. While the secure memory execution model introduces around 10% to S-App execution (as in [4]), it tends to introduce large performance degradation to co-run applications. When there are multiple channels, dummy messages are generated to hide the access pattern. Since these messages are executed in parallel, they have less impact on S-App but degrade co-run NS-Apps significantly.

4.2 THE MEMORY SYSTEM ARCHITECTURE

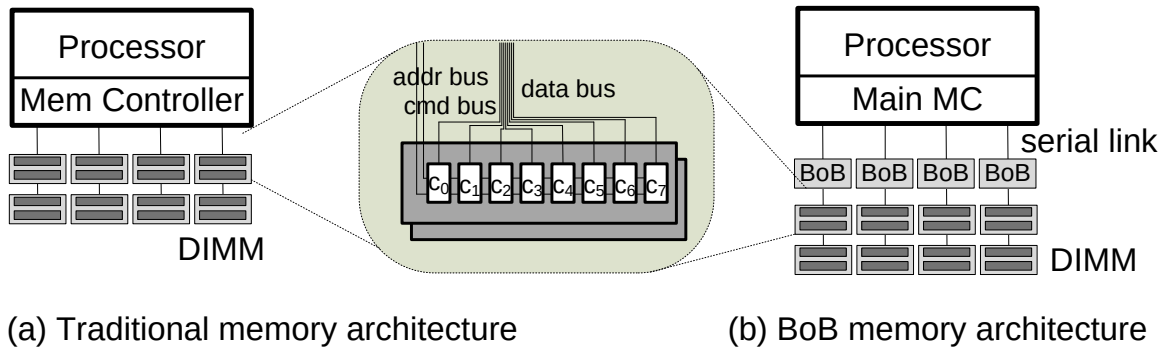


Figure 30: The DRAM based memory system architectures.

The DRAM based memory system traditionally adopts the direct-attached memory architecture, as shown in Figure 30(a). One memory channel connects to one or more DRAM DIMMs while each DIMM consists of two memory ranks and one rank consists of eight (no ECC) or nine (with ECC) DRAM chips. The channel bus consists of address, data, and command buses. While the address and command buses link all chips using, e.g., daisy-chain, the data buses from each chip are aggregated to form the channel data bus.

An on-chip memory controller (MC) is integrated on the processor. When servicing a memory read or write request, the MC sends out a sequence of device commands, e.g., precharge, activate, read, or write, to operate the memory chips. The time intervals between

device commands are specified by JEDEC standard [3]. Different memory channels may be ganged together, i.e., operate synchronously, to form wide data buses and improve the system bandwidth for high throughput applications.

To address the capacity and bandwidth demands of modern computing servers, recent memory architectures place memory buffers (and their associated logic) between CPU and DRAM chips, ranging from a simple buffer that re-drives the signal to boost signal integrity [42], to a buffer-on-board (BoB) unit that controls the DRAM and receives requests and sends data back to the processor [14], to the HMC architecture [56] that adopts 2.5D/3D integration to have control logic as well as other simple operations (e.g., ECC and cryptographic operations) in the logic layer on top of memory chips. The last two designs communicate with the processor using narrow but fast serial link buses — the requests being sent to and the responses from the DRAM chips are encapsulated as data packets.

While BoB and HMC architectures share many similarities, there is a significant difference from security enhancement point of view, i.e., the buses between BoB buffer and DRAM chips are visible to attackers while the buses between HMC and DRAM subarrays are embedded inside the HMC module. Therefore, the DIMMs are still untrusted in BoB architecture — it is possible to attach physical devices to tamper with the communication [43, 77].

BoB architecture not only supports large capacity memory but also is compatible with commodity DIMMs. It has better adoption in mainstream servers than HMC. IBM power8 supports eight memory buffers with each controlling 128GB memory and 1TB per socket [67]. Oracle M7 supports up to 16 DIMMs using eight BoB buffers and 1TB per processor [33]. Intel Xeon E7 [34] adopts proprietary Scalable Memory Buffers (SMBs) that supports up to three DIMMs per buffer and 1.5TB per socket. While the SMB details are not released to the public, SMB controls DIMMs only and thus is similar to BoB rather than Fully-buffered DIMM [38]. As a comparison, HMC faces fabrication challenges for improving module capacity and TSV yield at present. The first processor that uses HMC was Fujitsu SPARC64 XIfx [25], which was released in 2015 and connects to 32GB memory using eight 4GB HMC modules.

4.2.1 Design goal

In this work, we are to devise a novel secure execution model that is compatible with mainstream server memory architectures, i.e., it prevents information leakage from memory accesses to untrusted memory. Our design goal is to achieve high-level security protection, high system resource utilization, low interference between secure and non-secure applications, and good compatibility with mainstream server hardware.

4.3 THE D-ORAM DESIGN DETAILS

In this section, we first present an overview of the proposed D-ORAM scheme and then elaborate the details and performance optimizations.

4.3.1 Overview

An overview of the D-ORAM memory system is illustrated in Figure 31. An 8-core CMP has four BoB based memory channels — each channel has a main BoB controller on the processor chip and a simple controller on the motherboard, i.e., $\texttt{MainMC}_i$ and $\texttt{SimpleMC}_i$, respectively ($0 \leq i \leq 3$). Residing between the processor and the commodity memory DIMMs, the simple controller contains both control logic and data buffers. BoB architecture uses the serial link to connect the main controller and the simple controller, and parallel link to connect the simple controller and the DIMMs. The simple controller is responsible for sending out device commands and enforcing the timing constraints as specified in JEDEC standard.

By default, D-ORAM upgrades one memory channel, i.e., channel-0 in the figure, to *secure channel* which delegates Path ORAM. Other channels are *normal channels*. Multiple memory channels can be upgraded for higher design cost.

For discussion purpose, the peak bandwidth of one serial link channel is set to be comparable with that of one parallel link channel. Each simple controller controls one to four sub-channels. There are two reasons: (1) we are to compare with the direct-attached memory

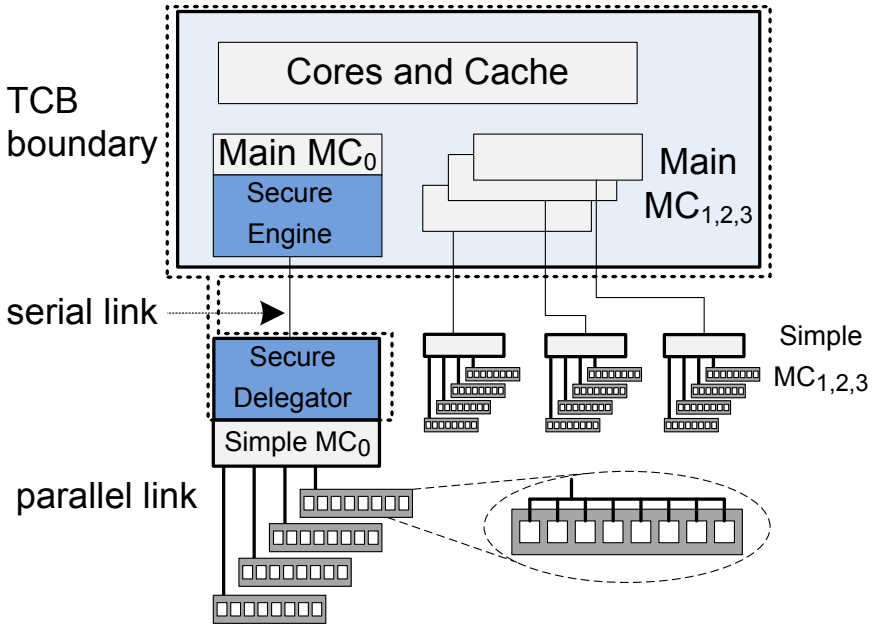


Figure 31: An overview of D-ORAM memory system.

architecture that uses four parallel link channels. The two settings have comparable peak off-chip memory bandwidth. (2) we are to study the space advantage of BoB architecture to support large capacity memory systems.

D-ORAM introduces an extra secure component, referred to as **secure delegator (SD)**, to the on-board simple controller, which not only accelerates cryptographic operations but also enforces Path ORAM.

TCB=CPU+SD. The TCB in D-ORAM includes both the processor (CPU) and the secure delegator (SD). SD has two responsibilities:

(1) It secures the communication between CPU and SD. The sender encrypts and adds authentication and integrity check bits before sending out the message while the receiver decrypts, authenticates and integrity checks before use.

(2) It performs Path ORAM accesses to the untrusted memory. In particular, it converts one memory request from the processor to hundreds of memory accesses to a path on the ORAM tree.

A major difference between the proposed D-ORAM model and the secure memory model is that D-ORAM does not include memory modules in the TCB, that is, even though SD is physically integrated with the on-board BoB unit, the BoB components, e.g., the controller logic and the queue buffer do not need protection. Thus, the address and command buses (that connect the simple controller and memory modules) transmit cleartext data that are visible to attackers. Such a setting matches the wide adoption of untrusted commodity DIMMs in server settings. The commodity DIMMs need cleartext addresses and device commands with timing following the JEDEC standard. Our design does not need to redesign the DRAM interface and device.

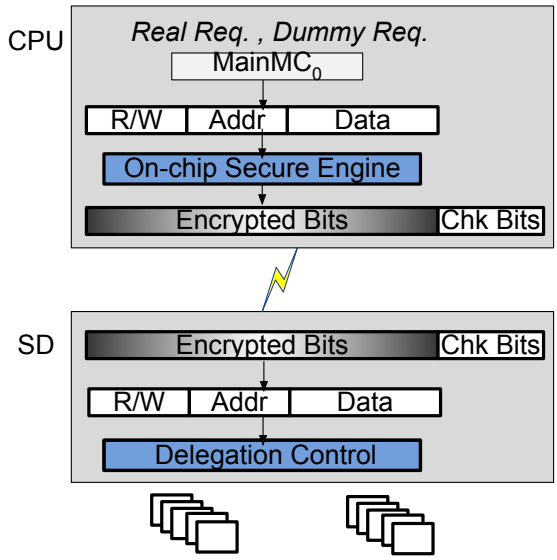


Figure 32: Delegating Path ORAM in SD.

4.3.2 Path ORAM delegation in SD

We first present how SD protects the memory accesses in D-ORAM. Intuitively, SD protects the communication between CPU and SD through an encrypted channel, similar to that in InvisiMem; and the communication between SD and DIMMs using Path ORAM.

Let us assume the system is running one S-App and one or multiple NS-Apps. The OS allocates space from all four channels to the NS-Apps and space from the secure channel to

the S-App. In particular, S-App builds the Path tree covering 4GB memory space. Each tree node contains four blocks (i.e., cache lines) that are distributed to four sub-channels controlled by $\texttt{MainMC}_0$. D-ORAM works as follows.

The processor triggers SD for operation. In D-ORAM, the SD unit is triggered by a memory request sent from $\texttt{MainMC}_0$. When S-App encounters a cache miss and needs to access the main memory, $\texttt{MainMC}_0$ prepares a BoB packet, as shown in Figure 32. Each packet is 72B long, which includes three fields: access type (1 bit, i.e., read or write), memory address (63 bits), and data (512 bits). D-ORAM enhances the baseline packet preparation in BoB scheme to prevent information leakage.

- D-ORAM always attaches a 64B data field to the packet such that a read request is non-distinguishable from a write request. This helps to prevent potential information leakage from request types [1, 4]. For the read requests, the data field contains dummy data, e.g., all 0s.
- D-ORAM, in addition to a real memory request from the secure application, may generate dummy requests to prevent timing channel attack [50, 24].

  In D-ORAM, the on-chip secure engine generates a new Path ORAM request t cycles after receiving the response packet of the preceding request. We choose t=50 in this work. If there is no real request from S-App, a dummy request is generated and sent.
- D-ORAM adopts the pseudo OTP (one-time-pad) encryption. Before program execution, the on-chip secure engine and the SD negotiate a secret key K and a nounce N_0. This can be accomplished by adopting the public key infrastructure (PKI) as shown in [1].

  The on-chip secure engine generates a 72B-long OTP using AES encryption, and XOR the OTP and the packet as follows.

$$\begin{aligned} OTP &= AES(K, N_0, SeqNum) \\ SeqNum &= SeqNum + 1 \\ Enc\_Packet &= OTP \oplus Cleartext\_Packet \end{aligned} \tag{4.1}$$

  The $SeqNum$ is the message sequence number and is reset before execution. From the equation, it is clear that the OTP is not data dependent on the content of the transmitted

packet and thus can be pre-generated. Processing one Path ORAM takes long time (to finish hundreds of memory accesses to the Path tree) while it only requests two $OTPs$ for processor/SD communication — one for sending the request and the other one for receiving the response. The overhead is negligible.

- For high level security, the packet may need authentication and integrity check. The former prevents the attacker from injecting malicious packets. The latter prevents the attacker from replaying old packets. We adopt the similar designs in previous studies [4].

SD delegates Path ORAM. When SD receives the encrypted packet from $\texttt{MainMC}_0$, it decrypts and checks the data before processing it. SD then follows Path ORAM protocol to access the data saved in the insecure sub-channels. SD contains all the components that are necessary for Path ORAM (Figure 4). We will evaluate its hardware overhead in Section 4.3.5.

The processing follows the Path ORAM protocol [72]. It consists of the following steps: SD first consults the position map to locate the path along which the requested data is saved; it then reads all data blocks from the path; it remaps the requested block to another path and has it saved in the stash; it re-encrypts other blocks along the path and write them back. The blocks in the stash, if can be combined, are written back as well. As discussed, one tree node consists of four blocks that are distributed to four sub-channels. All four sub-channels are accessed in parallel to minimize the performance degradation.

SD returns the response packet. When SD finishes the read phase, it prepares a 72B-long response packet. The data field contains the dummy bits if the request from the processor was a write request. The response packet needs to be encrypted and has checked bits added before being sent back to the processor chip. The processor chip checks the packet and decrypts it to get the requested data for the read request or finish for the write request.

Timing control in D-ORAM. To prevent the timing channel attack, the processor chip sends out a new request is t cycles after receiving the response packet. In this work, we set t=50. If the write phase is ongoing when the processor chip receives the response packet, the new request is buffered in the SD and will be serviced after the write phase for the current request. The simple controller is responsible for generating the detailed device commands and device access time to the insecure memory.

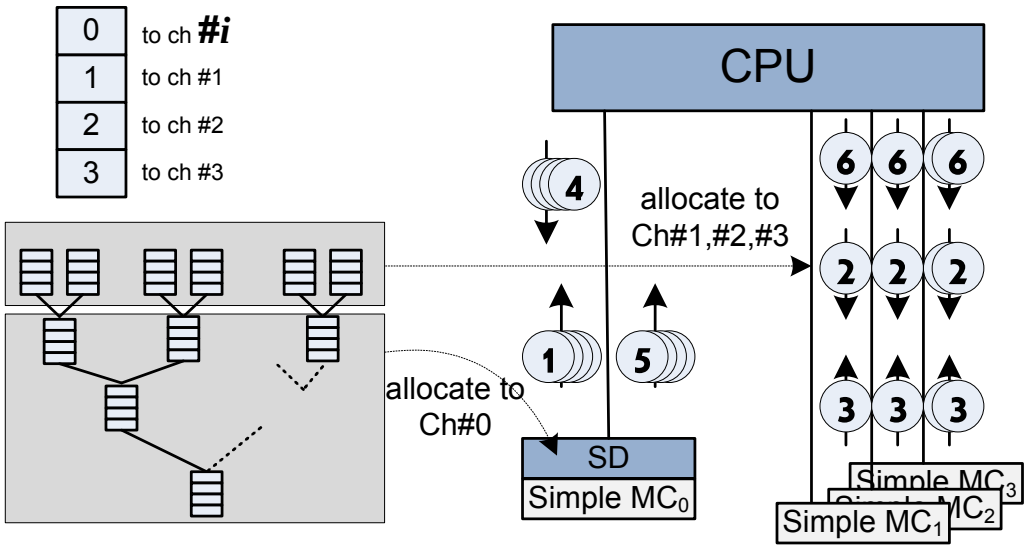


Figure 33: Splitting a Path ORAM tree access across channels.

4.3.3 Expanding Path ORAM tree across memory channels

In the default D-ORAM configuration, the secure channel consists of four sub-channels, which provides roughly the same bandwidth for S-App as the setting in previous Path ORAM studies [61, 80, 60], i.e., the one adopting four on-chip memory controllers for four parallel channels. In either setting, Path ORAM can utilize close to the peak memory bandwidth of each channel or sub-channel.

However, the default configuration may potentially run into space allocation problem. To prevent tree path overflow, a critical exception that fails the protocol, Path ORAM sets the space efficiency to be around 50% [72]. That is, a 4GB tree needs to be built for 2GB user data. In addition, when running, e.g., two S-Apps and two NS-Apps in D-ORAM, the two NS-Apps could have their data spread across all four channels but the two S-Apps allocate all their data all in the secure channel. Therefore, the secure channel tends to be under memory capacity pressure.

We then propose to balance the space demand by expanding the Path ORAM tree across channels. As shown in Figure 33, we observe that the nodes in the last level of the

Table 4: Balance space demand across channels (k>=1, $m \in$[k,2k])

k	Data Block Distribution		Extra Messages	
	channel #0	channel #1 to #3	channel #0	channel #1 to #3
1	50.0%	16.7%	4k **short** Read packets,	m **short** Read packets,
2	25.0%	25.0%	4k Response packets,	m Response packets,
3	12.5%	29.2%	4k Write packets	m Write packets

tree account for around 50% space — there are 2^L nodes in level L and (2^L-1) nodes in total from level 0 to level L-1. Let us denote the two sets as S1 and S2, respectively. Given one path that contains L+1 nodes including the root node (level 0), we access 1 node from set S1 and L nodes from set S2.

Based on the imbalanced accesses to the tree node sets, we propose to relocate the last k levels to other channels to balance the space demand across the channel. Since each tree node contains four data blocks, we distribute them to channels #i, #1, #2, and #3, respectively, and $i = (path\_id \ mod \ 3)+1$. That is, the nodes have their first blocks alternatively allocated in three normal channels. As shown in Figure 33. Table 4 compares the percentages of the blocks saved in each channel when splitting the last k levels into normal memory channels. For example, when k=2, each channel saves 25% data blocks of the path tree.

To conduct Path ORAM protocol under the optimized data allocation, the SD and the on-board simple controller send out explicit requests to access the k nodes (or $4k$ data blocks) from the last k levels. For simplicity, SD sends out $4k$ read packets to explicitly ask for the blocks from the other three normal channels. [1] Here, the read packets are short packets with data field omitted. This is safe because the optimization is well known such that the message types at this step are also known to attackers. The response packets are of 72B each. The fetched blocks are first returned to the on-chip memory controller and then forwarded to the SD in the secure channel. The data blocks are then updated during the write phase with Write requests sent from the SD and forwarded by the main controllers.

[1]Some read packets may be merged, we leave it to the future work.

An interesting property of this optimization is that there is no need to upgrade the normal channels. Given that the contents saved in the path tree are encrypted and optionally authenticated for higher level security, the normal channels cannot derive private information from the access. Neither the read request packet nor the response packet demand additional encryption — the read packet can be sent in cleartext while the response packet contains the fetched data (already ciphertext) from the memory.

A disadvantage of the design is that it overburdens the serial links with extra messages. Table 4 compares the number of extra messages with different k values. From the table, when k=2, D-ORAM sends 8 extra short read packets to CPU and 8 response packets to SD on channel #0; and 2 to 4 read and response packets on each normal channel.

4.3.4 Secure channel sharing

The secure channel in D-ORAM, comparing to those normal channels, tends to be overloaded. Channel#0 services not only S-App but also NS-Apps. Given that S-App is extremely memory intensive, there exists significant contention between S-App and NS-Apps for the sub-channels. Our study showed that even adopting the cooperative Path ORAM optimization [80], the performance degradation to memory accesses of this channel cannot be ignored. In addition, adopting the space demand optimization introduces extra messages. When k=2, the secure channel suffers from 24 extra messages while each of normal channel has 6 to 12 extra messages.

Figure 34 illustrates the memory access latency when we allocate NS-Apps using different memory channels. Figure 34(a) and (b) show that when there is no S-App, the channel access latency is longer when there are fewer channels. While our proposed technique can eliminate the interference in non-secure channels, as shown in Figure 34 (c), the secure channel is still slower than other channels.

Given one S-App and multiple NS-Apps, we profile the performance of three co-run settings and estimate the channel contention. For each setting, we first compute the slowdown of the *average memory access latency* on NS-App, i.e., the latency degradation of co-run setting over the solo run setting; and then compute the average slowdown of all NS-Apps.

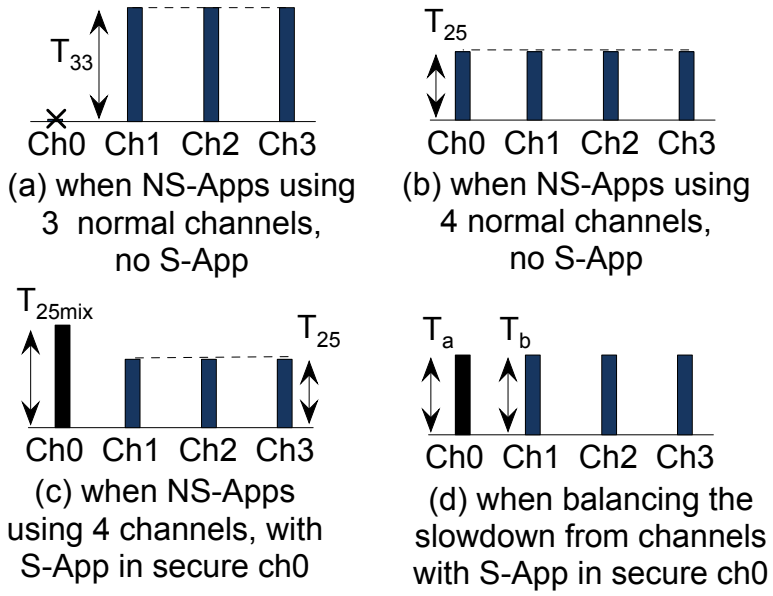


Figure 34: Balance the average access latency between secure and normal channels.

We have:

(1) T_{33} is the average memory access latency slowdown when NS-Apps use the three normal channels but not the secure channel, i.e., each channel has 33% traffic;

(2) T_{25} is the average memory access latency slowdown when NS-Apps use all four channels but the S-App is not active, i.e., each channel has 25% traffic;

(3) T_{25mix} is the average memory access latency slowdown when NS-Apps use all four channels and S-App uses the secure channel. T_a and T_b are the average memory access latency of secure and normal channels after balancing.

Our goal is shown in Figure 34(d), which achieves similar channel access latency T_a and T_b by adjusting the memory traffic to Channel #0.

We propose to alleviate the contention on the secure channel by adjusting the data allocation of NS-Apps and directing fewer NS-App requests to this channel. Our technique is to adjust the number of NS-Apps that can use the secure channel. By default, all NS-Apps can allocate memory from channel #0.

By reducing the number of NS-Apps that can use the secure channel, Channel # 0 shall become less congested. However, if most NS-Apps use normal channels, the overall performance is close to T_{33}, which may become sub-optimal due to bandwidth contention on normal channels.

We find the optimum allocation threshold by profiling application's channel access latency, T_{25mix} and T_{33}. We calculate the ratio of $r = T_{25mix}/\ T_{33}$, if $r > 1$, the secure channel is slow to handle more traffic from NS-App, and if $r < 1$, it is better to fully utilize all channels to handle traffic from NS-App. We show the profiling results and how the ratio impacts our threshold chosen in section 4.5.3 .

4.3.5 Overhead of secure delegator

We need to enhance the hardware on BoB to enable D-ORAM. The secure delegator embedded in the BoB unit is responsible for conducting Path ORAM operations. As shown [60], this secure component (including the stash, encryption logic, etc) occupies less than 1 mm^2 die area using 32nm technology node. This is modest for an on-board BoB unit.

4.3.6 Extension to parallel link buses

In this work, we utilize the BoB architecture to enable low-interference low-cost secure execution model on untrusted memory. Extending the design to parallel link bus based direct-attached architecture is possible but demands modifications to the channel organization. For example, if the data buses of individual memory chips are aggregated by an on-DIMM bridge chip, e.g., the UDIC controller in [18], it is possible to offload the secure delegator in the UDIC. That is, the TCB consists of the processor chip and the secure delegator in the UDIC. Such an extension demand timing adjustment to enable a non-blocking read operation. In summary, offloading to traditional direct-attached memory architecture is possible but tends to introduce higher overhead.

4.3.7 Security analysis

Our D-ORAM design focuses on reducing the application execution interference by delegating the secure engine to BoB unit. The protocol of Path ORAM remains unchanged, hence, the protection strength is not affected. The access pattern of S-App from the SD to DRAM DIMM is still protected by Path ORAM which does not show any information leakage through the plaintext on the conventional directed attached interface. Meanwhile, the data access on the serial link is sent via uniformed encrypted package, as discussed in Section 4.3.2. As long as the TCB boundary is not broken, i.e., the secure engines in the processor and BoB buffer (Figure 31) are not compromised, our proposed secure memory architecture can still provide S-App highest protection.

In our co-run model, we assume that multiple applications sharing the same memory bandwidth. The NS-Apps are not considered as malicious in such model. In the case that NS-Apps are spy programs, other types of side channel attacks may be launched, such as timing channel attacks described in Section 2.2. By fixing memory access rate for S-App, memory-level timing side channel can be prevented, as studied in previous ORAM optimization work[95, 24]. Additional countermeasures for cache level attacks may be added to the system cache protocol, which is beyond the scope of the focus of the work, which is addressing the potential leakage access pattern on the main memory bus.

4.4 EXPERIMENTAL METHODOLOGY

To evaluate the proposed D-ORAM scheme, we simulated an 8-core CMP with four off-chip memory channels. We used USIMM, a cycle accurate memory system simulator with processor ROB front-end[8]. We modified the default DDR memory interface to simulate the proposed architecture and compare them to the state-of-the-art. We added 15ns data transfer latency for the overhead of link bus and BoB control.

Table 5 summarizes the baseline processor and memory configuration. The DRAM memory follows JEDEC DDR3-1600 specification. We adopted the default values in the specifica-

tion that are strictly enforced in USIMM. Each application has its own memory space. The baseline S-App Path ORAM tree occupies 4GB memory space. The Path ORAM configuration is: $L = 23$, $Z = 4$. We used tree-top cache to cache top three levels of nodes, and the rest of 21 levels are divided into three section of 7-level subtrees, in order to maximize the row buffer hit rate[61]. Each memory channel can consist of one to four sub-channels. We choose to set the secure channel with 4 sub-channels, and other channels with 1 sub-channel, in order to fairly compare with previous techniques.

When S-App and NS-App are sharing the same memory channel, we adopt the bandwidth preallocation technique in [80]. We set the threshold to 50% so that both kinds of applications have similar slowdown.

Table 5: Baseline System Configuration

Parameter	Value
Processor	8-core, 3.2GHz
Processor ROB size	128
Processor retire width	4
Processor fetch width	4
Last Level Cache	4MB
Memory Device	DDR3-1600
Memory channels	4
Sub-channel per channel	1-4
Rank per Sub-Channel	1
Bank per Rank	8
Buffer Logic and Link latency	15ns[16]

We choose 15 memory intensive benchmarks used in MSC [40]. These benchmarks are chosen from PARSEC suite, commercial and BioBench. Each benchmark trace consists of 500 million representative instructions out of 5 billion instructions, using a methodology similar to Simpoint [40]. The workloads used for evaluation consists of one S-App and seven NS-Apps: S-App version adopts encryption and Path ORAM protection (or other protection

Table 6: Simulated Benchmark Programs

Suite	Workloads
PARSEC	black (4.2), face (26.8), ferret (8.0), fluid (17.5), stream (12.9), swapt (10.9)
COMM.	comm1 (7.3), comm2 (12.6), comm3 (4.2), comm4 (3.7), comm5 (4.5)
SPEC	leslie (23.1), libq (12.0)
BIOBENCH	mummer (24.0), tigr (6.7)

schemes) while NS-App version does not. The addresses of different versions are mapped to different address spaces. Our results use the same program for S-App and NS-App.

Table 6 summarizes the benchmark programs with their corresponding MPKI (memory access per kilo instructions) listed in parenthesis. We used the first two letters of each program to indicate the workload in the result section.

4.5 RESULTS

In the experiments, we evaluated the following schemes:

- `Baseline`. This is the baseline architecture without modified memory interface. It uses 4-channel direct-attached DRAM interface to run the mixed S-App and NS-App workloads. For S-App, the ORAM accesses are evenly distributed into each channel as previous work [80]. The results of other schemes are normalized to `Baseline`.
- `D-ORAM`. This scheme implements the proposed BoB based memory architecture, with the secure delegator (SD) on channel #0. It does not apply either space or channel

optimization. The S-App is mapped to Channel #0 with its addresses being allocated interleavingly across four sub-channels. The NS-Apps are mapped to all four channels with their addresses being allocated interleavingly across four channels.

- `D-ORAM+k`. This scheme is built on top of `D-ORAM`. It allows S-App to use other channels while the SD still stays with Channel #0. We modeled the memory communication across channels. k denotes that the number of extra tree levels that the Path ORAM tree expand. The tree space doubles when k=1.
- `D-ORAM/c`. This scheme is built on top of `D-ORAM`. This technique controls how NS-App can utilize channel #0. Parameter c means the number of NS-Apps that can use channel #0. In our setting, c can vary between 0 to 7. `D-ORAM/7` is the same as `D-ORAM`.
- `D-ORAM+k/c`. This scheme combines `D-ORAM+k` and `D-ORAM/c` to illustrate the effectiveness of channel sharing under tree expansion.

4.5.1 Performance evaluation

We first analyzed the performance under different protection settings. Figure 35 shows the normalized execution time of `Baseline`, `D-ORAM`, `D-ORAM/X`, `D-ORAM+1`, and `D-ORAM+1/4`. Here, `D-ORAM/X` means the best result can be achieved by varying the parameter c from 0 to 7. The detailed bandwidth sharing results can be found in the following section.

From the figure, we observed that `D-ORAM` reduces the execution time to 87.5% of `Baseline`. The reduction mainly comes from utilization of fast non-secure channels. However, the secure channel is still shared by all NS-Apps and S-App. By adjusting the number of cores using the secure channel, the execution time can be reduced further to 77.5% of baseline, representing 22.5% performance improvement by using `D-ORAM/X`.

Our technique allows large Path ORAM tree storage across the secure channel and other channels. In the figure, `D-ORAM+1`, the one that allocates all leaf nodes to other three non-secure channels, only slightly slower than `D-ORAM`. We observed that, on average, the execution time is 88.6% of `Baseline`. Adopting the bandwidth sharing technique, for example, when allowing 4 NS-Apps (`D-ORAM+1/4`) to use the secure channel, the execution time can be reduced to 81.4% of `Baseline`.

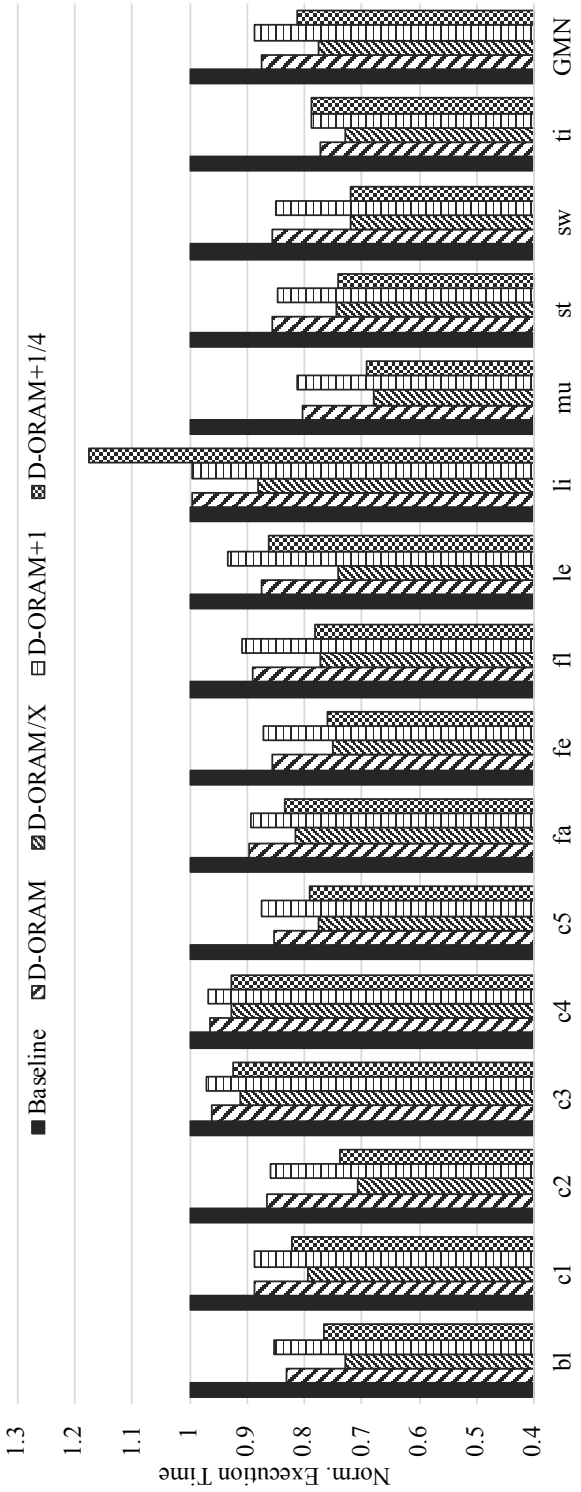


Figure 35: Comparing NS-App performance when adopting different D-ORAM schemes

4.5.2 Expanding the Path ORAM tree

Figure 36 shows the performance impact of space expansion. We varied the `k` from 1 to 3, meaning that we added extra `k` levels to the original 4GB Path ORAM tree and the capacity of Path ORAM tree expands from 4GB to 4×2^kGB.

The introduced overhead to NS-App is minimal. Compared to the D-ORAM, varying k from 1 to 3 adds additional 1.02%, 2.01%, 3.29% execution time. This is because that the extra memory accesses introduced by channel communication are not significant. For the secure channel, the extra traffic is limited between processor and BoB controller. For other channels, because $k \times 4$ blocks are distributed to 3 channels, the impact is also not significant.

4.5.3 Secure channel sharing

We then studied the effectiveness of secure channel sharing. Figure 37compares the performance under different D-ORAM settings. In particular, we compared the performance when allowing 0 to 7 NS-Apps to utilize the secure channel, i.e., having their data allocated to the secure channel. We included the results of 7NS-3ch and 7NS-4ch for comparison.

From the figure, we observed that different applications prefer different channel sharing configurations. To determine the optimal setting for different applications, we found that the two parameters, T_{25mix} and T_{33}, are critical for identifying the best sharing configuration. We use a different segment of memory trace as profiling input and then compute the T_{25mix}/T_{33} ratio, as shown in Figure 38. We show that our simple ratio calculation can guide the program to choose the optimum c setting.

When the ratio is bigger than 1, i.e., $T_{25mix} > T_{33}$, we prefer to let fewer NS-App copies to use all four channels, e.g., *bl*, *cx* and *mu*. Therefore, c should be set to a smaller number in this case. When the ratio is smaller than 1, we prefer to let more (e.g., 5 to all) NS-App copies to use all four channels, e.g., *le*, *li*, *st* and *ti*. In Figure 38, there is only one exception *c*2, which has best configuration $c = 1$ in experiment but falls on the other side of the figure. We believe that this is because the ratio is very close to 1. For other benchmarks, our profiling guidance works in accordance with the best parameter we achieved in experiments.

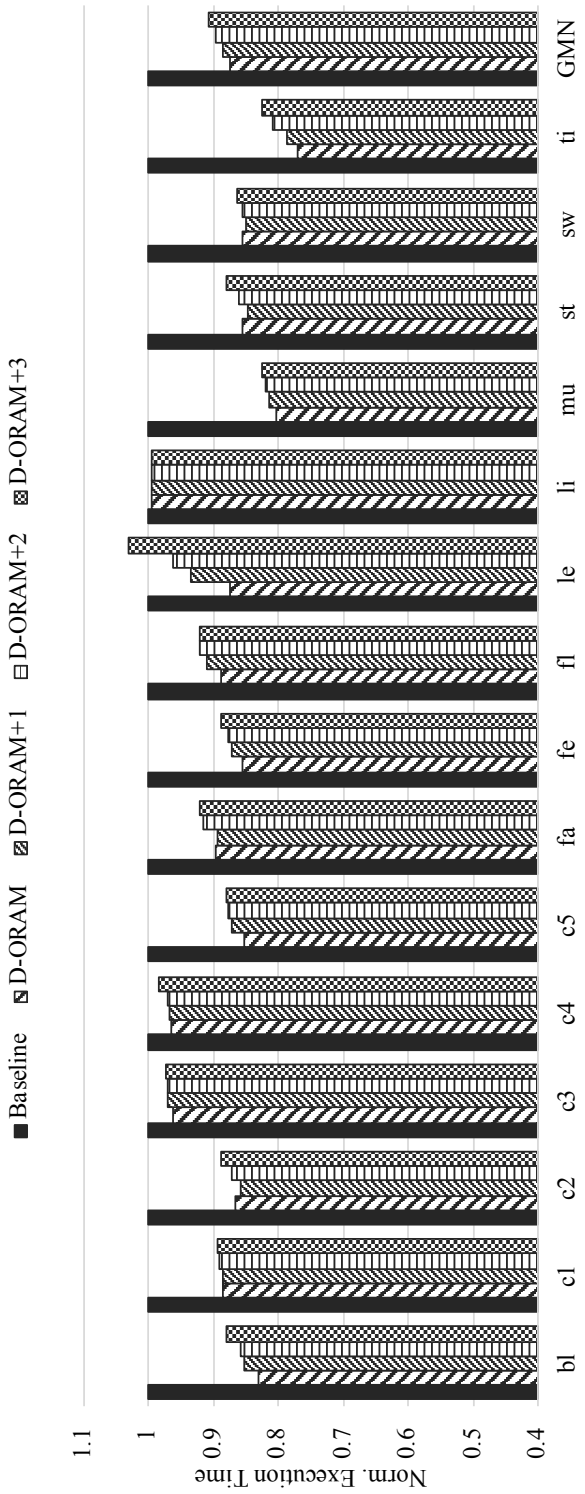


Figure 36: Comparing the performance impact when using large Path ORAM trees.

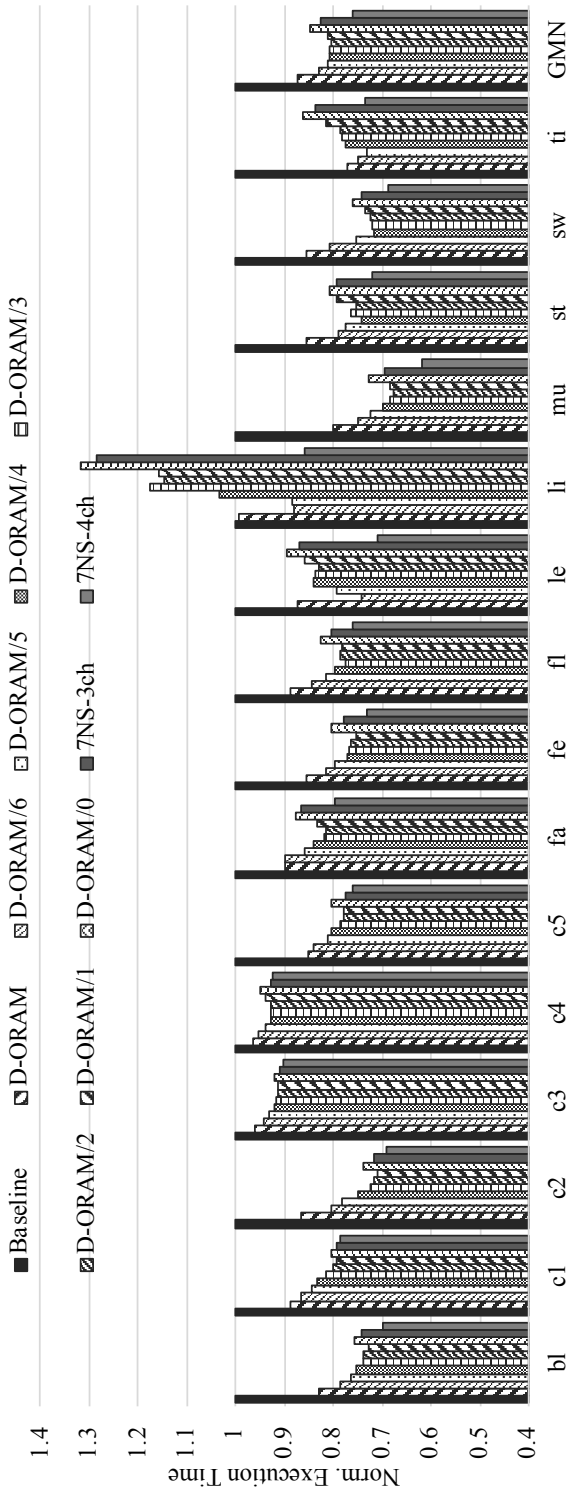


Figure 37: The performance impact when adopting secure channel sharing.

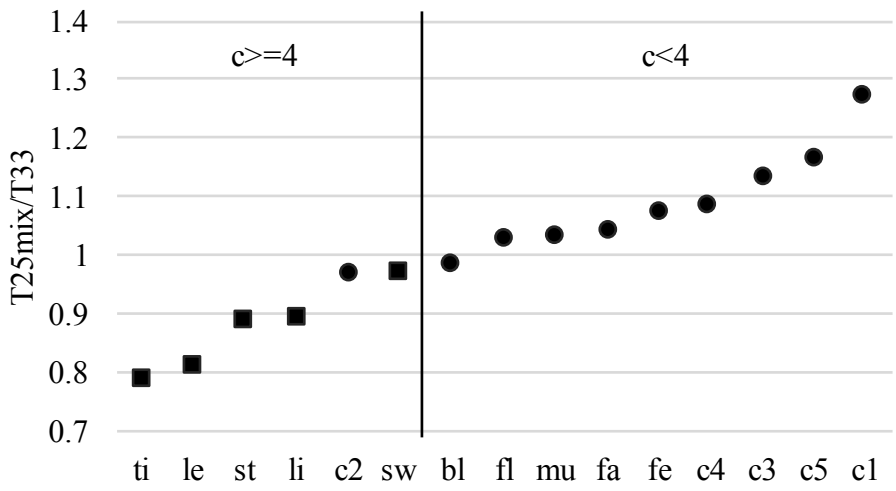


Figure 38: The ratio of T25mix and T33

4.5.4 Access latency reduction

We also compared the average NS-App access latency reduction in Figure 39. In this experiment, for illustration purpose, we chose `D-ORAM+1` and `D-ORAM/4`for the space expansion and secure channel sharing optimizations, respectively. On average, the NS-App read access time can be reduced to around 70% of the baseline. The write access time can be reduced to 48% of the baseline.

4.5.5 The performance impact on S-App

D-ORAM was designed primarily for improving NS-App performance and maps S-App mapping to a secure channel. In D-ORAM design, adopting Secure Delegator in BoB architecture slows down the memory access latency by tens nanoseconds. However, Path ORAM accesses typically finish in the range of thousands of nanoseconds [4, 61]. The overhead from BoB architecture is small.

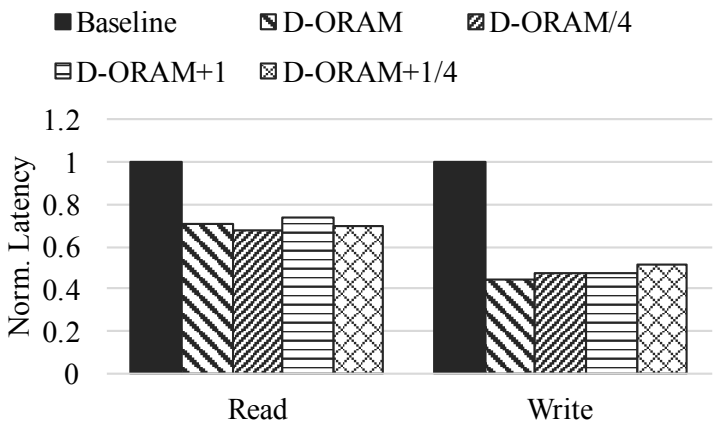


Figure 39: Comparing NS-App memory access latency.

4.6 CONCLUSIONS

In this chapter, we propose D-ORAM, a secure memory system that minimize execution interference on cloud servers. D-ORAM propose to utilize the buffer-on-board (BoB) like memory architecture to offload the Path ORAM operations to a secure engine in the BoB buffer, which greatly alleviates the contention for the offchip memory bus between secure and non-secure applications. Our design upgrades only one secure memory channel, and enables Path ORAM tree split to extend the secure application flexibly across multiple channels, in particular, the non-secure channels. We propose secure channel bandwidth sharing which further improve the system performance. Our evaluation shows that D-ORAM effectively protects application privacy on mainstream computing servers with untrusted memory, with an improvement of NS-App performance by 22.5% on average over the Path ORAM baseline.

5.0 RELATED WORK

5.1 SECURE PROCESSOR DESIGN

The XOM processor is one of the first secure processor designs [44]. XOM-based processors focus on protecting data secrecy [73, 86, 74, 52]. Later studies defended potential information leakage from address and command buses [96]. Studies showed that completely defending information leakage demands ORAM [27]. Path ORAM [72] is a simple and practical ORAM protocol that received wide adoption. The first hardware implementation of Path ORAM was Phantom [50] based on FPGA. Intel SGX[15] is the current state of art commercial and open-source secure processor available on the market, which protect sensitive data with encryption, integrity check in reserved EPC memory. AMD SME/SEV[41] provide similar secure computation features such as memory encryption and encrypted visualization.

5.2 SECURE MEMORY ARCHITECTURES

Each different secure memory architectures have their advantages and specific use cases. The recent proposed ObfusMem [4] and InvisiMem [1] schemes assume that memory is secure, which can be exploited to significantly reduce protection overhead. ObfusMem [4] assumes that the computation logic on the memory DIMM is capable to encrypt and decrypt the address, therefore, ORAM is not needed for expensive address obfuscation. InvisiMem [1] assumes an HMC like interface, where the secure engine can be placed inside of the HMC logic die. Motivated by near data processing [5, 6], Gundu et.al propose to use a secure DIMM [29] for integrity verification. They proposed to put a bridge chip on memory DIMM

that can handle merkle tree verification, and reduce the memory traffic between processor and memory. More recently, Shafiee *et al.* [65] proposed to utilize buffer chip on DRAM DIMM to reduce ORAM traffic, which requires redesign the memory access timing protocol.

5.3 ORAM OPTIMIZATIONS

The large performance overhead of ORAM has been a focus of recent ORAM designs. Ring ORAM[59], Bucket ORAM[22] were proposed to reduce the bandwidth overhead on the memory bus by using different bucket organization and more complicated access flow control. This dissertation mainly focus on Path ORAM, and most of the designs can still apply to different ORAM schemes, especially tree based ORAMs. There are also many design challenges when we use new ORAM as the secure primitive, which are discussed in the future work section.

To improve Path ORAM performance on DRAM based system, several techniques have been proposed. Ren *et al.* [61] optimized block mapping using sub-tree layout, which maximizes row buffer hit for ORAM accesses. They saved the top of the Path ORAM tree in a small on-chip cache to improve performance. Zhang *et al.* [95] eliminated unnecessary memory accesses if consecutive path accesses have overlaps. An ORAM prefetcher [92] is proposed to improve the ORAM path access locality and performance. Fletcher *et al.* [23] shows how to effectively reduce recursive ORAM overhead by caching the position map look up process.

5.4 NEW COVERT AND SIDE CHANNEL ATTACKS

Numerous of new covert and side channel attacks emerge in recent years, as well as tools and frameworks to detect such attacks. Chen *et al.* [9] proposed a framework CC-hunter that can detects the possible covert timing channels on shared hardware. The model assumes that the Trojan is able to intentionally communicate the secret to spy via covert channel. Hunter *et*

al.[32] used information theory to quantify the communication capacity of microarchitectural covert channels, and introduced a detection technique for covert channel eavesdropping attacks. Liu *et al.* [46] showed how an adversary can attack cross-core, cross-VM side channel and leak keys as well as secret data accesses via last level cache. Further, Liu *et al.* [45] proposed to defend such side channel attacks using a performance optimization cache allocation technology. Yan *et al.* [85] propose a secure hierarchy-aware cache replacement policy that defends against all existing cross-core shared-cache attacks with minimal hardware modifications. Yao *et al.* [88, 87] identified that there are covert channel information leakage via NUMA architecture and cache coherence protocol. In addition, they showed that hardware prefetchers can be used against to timing channel leakages [17]. Wang *et al.*[82] designed a queuing structure per security domain and allocated timing slots to different domains to eliminate timing channels. They further propose a trade-off between timing information leakage and performance[83].

This dissertation focus on the access pattern leakage on the level of main memory bus. Our work can be integrated with the cache attack protections schemes as needed, however, fully protect the system from all types of side channel attacks require sophisticated combination of all possible solutions.

5.5 NEW DRAM ARCHITECTURES

Recent studies proposed alternative memory architectures to alleviate the constraints in traditional memory systems. Fully Buffered DIMM [26] adds a buffer on memory DIMM to handles memory requests closer to the memory. While FB-DIMM can effectively increase memory density, it introduces high power consumption, making it a less popular architecture in modern servers. BOOM [91] adds buffer on DIMM to decouple internal and external buses, which achieves large power savings by matching the external high performance bus with multiple low performance internal buses. MIMS[10] replaces traditional bus interface with a universal message-based interface in memory system. The difference between MIMS and BoB design is that MIMS may improve communication efficiency by combining multiple

memory requests in one packet. Alloy[79] utilizes a heterogeneous memory interface, which includes DDR based parallel interface as well as serial interface, for improving performance on a GPU enabled heterogeneous system.

Our solution, D-ORAM also utilize some of the proposed DRAM interface, such as serial link interface and the concept of buffer in between of computation and storage. We further optimize the architecture to minimize ORAM applications interference and maximize system parallelism with the new architecture.

5.6 MEMORY SCHEDULING TECHNIQUES

Several memory scheduling algorithms have been proposed to achieve fair sharing of the memory bandwidth. Mutlu *et al.* [53] proposed to achieve fair schedule between streaming and random access applications in DRAM system. Mutlu *et al.* [54] improved fairness in memory scheduling using batching. Craeynest *et al.* [78] proposed equal-time scheduling and equal-progress scheduling to adjust the amount of resource that each thread receives.

However, these schemes did not consider the extreme biased co-run of S-App and NS-App, or the properties and guarantees of ORAM accesses. In comparison, part of this dissertation work introduced a fair and tunable memory scheduling solutions for faster memory response time for both type of applications.

6.0 FUTURE WORK

In this chapter, I summarize several potential projects that are related to this dissertation work. The target of such future work is to further enhance and enable ORAM to run in the server environment with low performance overhead and high usability.

6.1 ENHANCE RING ORAM OPERATIONS FOR FAST DATA RETRIEVAL

6.1.1 Problem Statement

A typical 4GB Path ORAM tree can be configured with `L=23` and `Z=4`. By spreading each bucket across all four memory channels, each Path ORAM read and write path operation can evenly touch all memory channel in balance.

Compared with Path ORAM, Ring ORAM reduced the online bandwidth by selectively reading one block from each bucket. Although the total touched blocks number is reduced, these blocks may come from different channels randomly. Among the read path operation, only one real block will be read out from a particular bucket, and all other bucket will provide one available dummy block.

From memory channel view, all blocks accessed during the read path can not be evenly distributed to four memory channels. Early reshuffle and eviction path operations will access all memory channels with evenly distributed memory traffic.

6.1.1.1 ORAM Tree Setting

We choose our settings for Ring ORAM tree has `Z = 8, A = 8, S = 12, L = 23` and each block is 64B, according to [59], this is one of optimum settings for small data blocks with negligible stash overflow probability. Therefore the Ring ORAM tree has a effective capacity of 8GB with total capacity of 20GB.

For simplicity, during one access, 24 blocks from 4 channels are sent to secure processor side. The ideal case would be that all 24 blocks are evenly distributed, however, in the baseline setting, it is almost impossible to make 4 channel finish at same time.

6.1.1.2 Prevent Timing Channel with Ring ORAM

To prevent timing channel leakage, one costly solution is to issue ORAM access with fixed rate[24, 21]. Followed by prior work, we define access rate r as the time from last access finish time to next access start time. Figure 40 shows how read path request can be serviced with timing protection. Request 1 and 2 arrive at memory controller during a request is performing, regardless of dummy or real, therefore it has to wait for current request finished and the access rate till it can be issued. Request 3 arrives during the wait period after Request 2, therefore it can be issued at next issue point.

To better utilize the memory channel, access rate r should be carefully chosen. If r is too small, more dummy requests need to be sent, which is a waste of energy. At the same time, it increase the probability that Request 1 in Figure 40 may happen. If r is too large, most of request will be serviced like Request 3. However, r itself will contribute most of wasted bandwidth time.

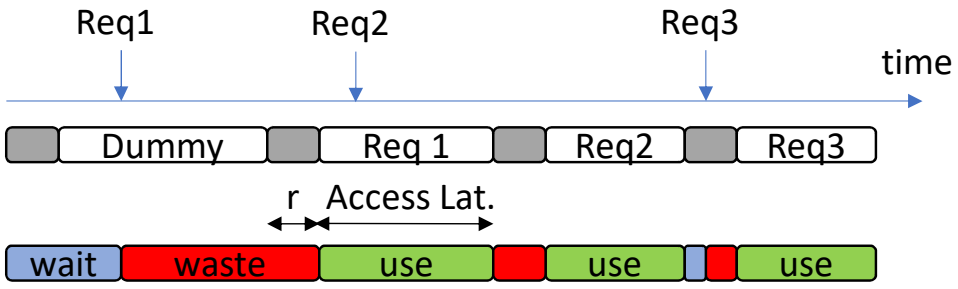


Figure 40: Protecting Ring ORAM from Timing Channel Attack

If we further look at the in-use time across channel, there are still wasted bandwidth, as shown in Figure 41. A normal read path access latency will be dominated by a channel that serves most actual read operations. The normal case in Figure 41 will prolong the access latency 66.67% over ideal case.

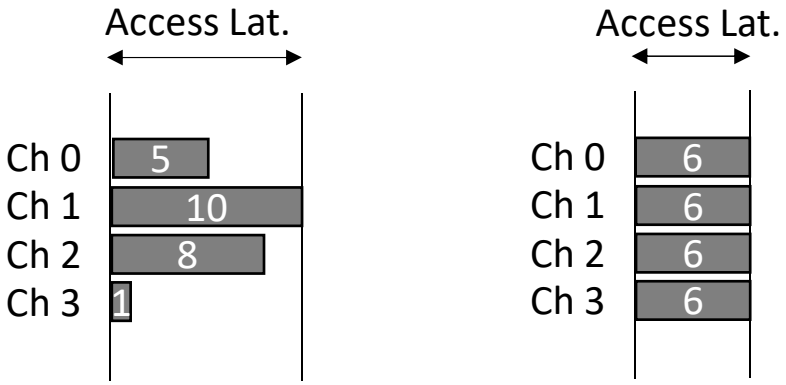


Figure 41: Ring ORAM Read Path Access Time, normal case v.s. ideal case

6.1.2 Balance Read Path to ORAM Access Latency.

The read path of Ring ORAM will return the block of interest and a number of dummy blocks from other buckets. The importance to balance read path is to identify whether we can select dummy block from each bucket in order to balance the overall access.

Before determine which block to fetch in each bucket, the ORAM controller need to read metadata from all buckets in the selected path. During the decryption, the ORAM controller will be able to identify real block and all available dummy blocks in each bucket. Therefore, before the actual read, the ORAM controller is able to which block can be selected from a bucket.

My proposed block selection workflow in the ORAM controller is as follow:

- Generate an initial block of selection in ORAM controller. Calculate the busy channel and idle channel of this access.
- Starting from leaf bucket to root bucket, if the bucket contributes a busy dummy block

and it has a valid dummy block from idle channel, switch current selected block with the dummy block from idle channel. Update the block of selection vector after each switch.

- After one round of dummy block switch, if the block of selection vector is now balanced, end the switch and start to fetch blocks from memory. If the channel is still imbalanced, continue to search dummy blocks to idle channel from buckets that do not contribute busy dummy blocks, until all channels are balanced.
- If the dummy blocks are not enough to balance this access, starting from leaf bucket to root bucket again and switch current selected block with an real block from idle channel. If the block of selection vector is balanced, end the search and start to fetch blocks from memory.

In very few cases, we will need to bring real blocks from idle channel to balance the traffic. This will increase the number of blocks in stash after an access. Increasing the stash size can reduce the overflow probability significantly. In addition, we propose Ring ORAM background eviction similar as Path ORAM background eviction [61].

6.1.2.1 Background Eviction Operation. When stash occupancy reaches a threshold $th = StashSize - Z * (L + 1)$, we stop issue real read path operation, as one more read path may. Instead, all blocks read during dummy read path operation can be discarded. During the Eviction operation, the content in stash is pushed back into the tree. When the stash occupancy decreased after the background eviction, the system can continue service real read path operation.

Service one background eviction is expensive. However, it rarely happens with large stash size.

6.1.2.2 Discussion on the XOR optimization In original Ring ORAM work, the authors mentioned that with computation on memory, if there is only one real block sent during a read path operation, they can apply XOR technique to reduce online bandwidth to 1. This optimization also requires that client set plaintext of all dummy blocks to a fixed value, such as 0. Therefore, on the controller side, the real data can be recovered by XOR ciphertext with encryptions of all dummy blocks.

In recent years, researchers propose that simple computation logic can be implemented on memory DIMM side such as [19, 30]. HMC and HBM provide even stronger computation logic layer that can do near data computing[37], but they do not use JEDE standard. Conservatively, we still consider that we have 4 conventional DDR DIMMs for 4 channel memory systems. In this case, we have to send at least 4 blocks, each from one DIMM, to the controller side, even with XOR optimization. Therefore, although we borrow real blocks from idle channel, if each memory channel has at most 1 real block, the XOR optimization can still be adopted.

In the extreme rare case that more than 1 real block is sent from 1 memory channel, the XOR optimization can not be used. Instead, we need to send all real and dummy blocks during the access to make the access indistinguishable. The attacker will learn that in this access, more than one real block is sent. This information does not hurt security, as the attacker can not learn which blocks are real or which channel has more than one real blocks.

6.1.3 Preliminary Results

In this experiment, I simulated 1 million Ring ORAM accesses with following ORAM tree configuration in Table 7, to test the effectiveness of block switching.

Table 7: Tested Ring ORAM Tree Organization

	Z	S	A
Case1	4	4	3
Case2	8	12	8
Case3	16	28	20

Figure 42 shows the normalized access latency reduction with tree height `L=23`. We can find that the worst case access time is independent from the tree bucket organization. By fixing the oram tree bucket setting as case 2, Varying L will give different improvement potential. Figure 43 shows that smaller L will introduce longer access latency in the baseline, as the imbalance is more significant.

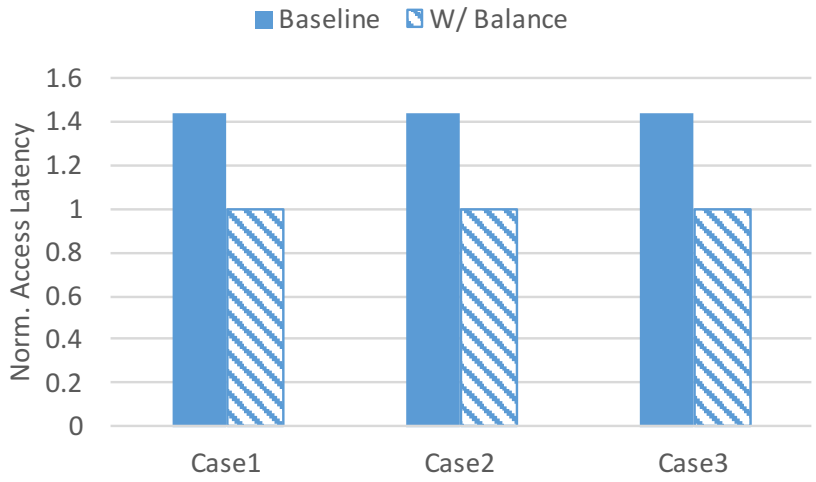


Figure 42: Access Latency Reduction After Channel Balance(L=23)

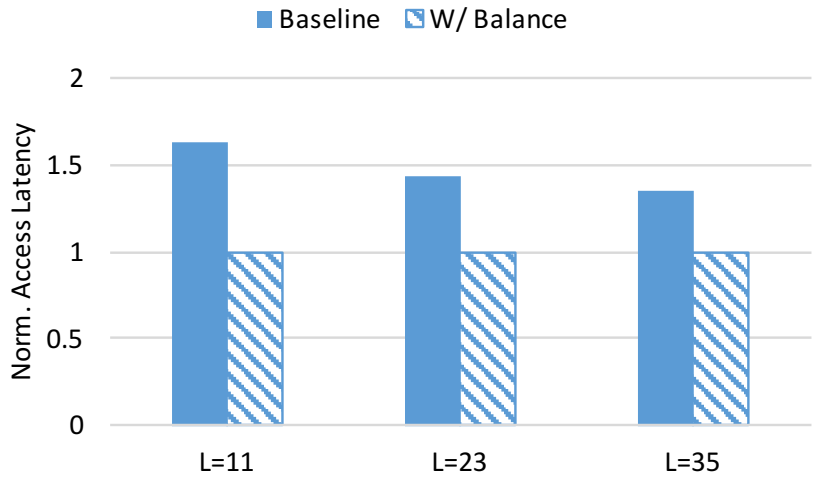


Figure 43: Access Latency Reduction After Channel Balance with various L(Case 2)

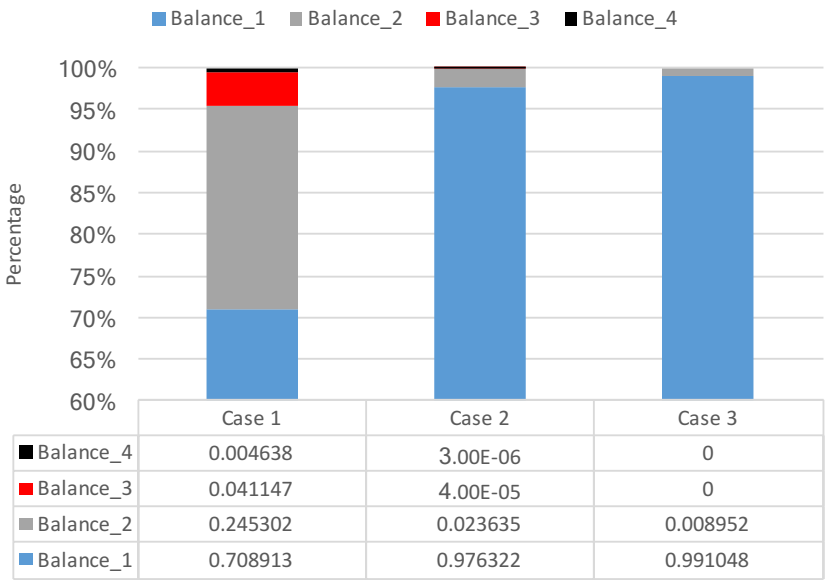
Figure 44 shows how the read path can be balanced through different types of block switch. Balance_1 refers to a bucket itself contributes a dummy block to busy channel, while it has idle channel dummy block available. By switching the block inside the bucket, a channel can get balanced.

Balance_2 is still another way of dummy block replacement policy. It treat the dummy block available in the path as a whole pool and try to find the dummy block from idle channel from a bucket that may not contribute to a busy channel traffic.

Balance_3 involves the real block replacement, however, it limit the maximum number of real block a channel can sent out to 1. It means that under such switch, the XOR optimization can still be used.

Balance_4 is the final choice for switching which do not limit the number of real blocks sent from a channel.

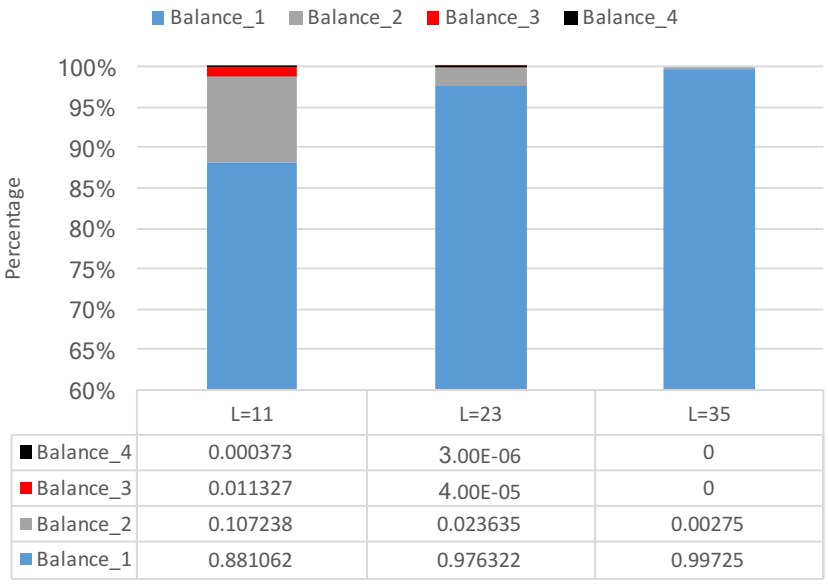
From the percentage breakdown, we can conclude that with larger number of blocks per bucket, a read path can be easier balanced with dummy block switch only. In case 1, when Z and S both are only 4, around 0.4% access requires balance4 scheme. When Z and S increase to 8 and 12, the percentage decreased to 10^{-6}.



	Case 1	Case 2	Case 3
Balance_4	0.004638	3.00E-06	0
Balance_3	0.041147	4.00E-05	0
Balance_2	0.245302	0.023635	0.008952
Balance_1	0.708913	0.976322	0.991048

Figure 44: Percentage of Access that can be balanced after schemes trial

Similarly, when we fix the bucket setting and vary the tree level, a larger tree will provide more dummy blocks to switch. When the tree height is small, for example `L=11`, the need for real block switch is around 1.1%, as shown in Figure45.



	L=11	L=23	L=35
Balance_4	0.000373	3.00E-06	0
Balance_3	0.011327	4.00E-05	0
Balance_2	0.107238	0.023635	0.00275
Balance_1	0.881062	0.976322	0.99725

Figure 45: Percentage of Access that can be balanced after schemes trial(Case2)

6.2 ENABLE ORAM TO USE HIGH DENSITY NVM

While adopting NVM in Path ORAM helps to enforce privacy protection on large data set with low leakage power, NVM faces intrinsic write performance and endurance issues, which demand further optimization for achieving overall system optimality. In this project, we will exploit the unique memory access characteristics of Path ORAM and its variants to improve NVM-based ORAM performance.

6.2.1 Exploit ORAM access patterns to prolong NVM lifetime.

Many NVM technologies, such as Flash, PCM, and some ReRAM variants, face write endurance problems. Given that ORAM primitives are write intensive, a major concern arising

from adopting NVM in Path ORAM implementation is, would the system be robust enough to meet the common server lifetime expectation, e.g., 5 or 8 years of service time?

In this project, we will investigate the hybrid integration of DRAM and NVM for Path ORAM, which can effectively reduce the number of writes to NVM and extend NVM lifetime. As an example, when evenly splitting ORAM space between DRAM and NVM, we may organize NVM as the last level of the Path ORAM tree. In this case, writing one ORAM path generates only one NVM write while tens of DRAM writes. Having data blocks from the last two levels allocated in NVM results in 75% of ORAM space being NVM and two NVM writes per path access. We will exploit Path tree structure to extend NVM lifetime. We will start with two approaches — reducing the number of bit changes and performing wear-leveling, and study other approaches during the course of the project.

For the purpose of reducing bit changes, we propose to use short memory rows, e.g., 32B instead of 64B. For an L-level path tree, one path access needs to read all data blocks of the path, re-encrypt them with different keys, and merge with blocks from stash, and write encrypted blocks back, resulting in modifying $L \times 4 \times 64B$ data. If using 32B block size, the tree would expand to $L+1$ levels such that we are to modify $(L+1) \times 4 \times 32B$, a large reduction of bit changes. However, using smaller block size results in large position map overhead, and increased cache misses. It would be more beneficial if the memory channel supports sub-channel operation. We will study these factors from which we identify the best tradeoff among bit changes, performance, and energy consumption.

We will then investigate techniques that exploit Path ORAM access patterns to extend the lifetime of the NVM subsystem. For example, memory cells are often accessed in rows while one row is usually bigger than the size of a cacheline, the typical data block size in Path ORAM. Studies have shown that subtree based organization, i.e., grouping consecutive nodes from their neighbors in one memory row, is optimum in maximizing row buffer hits. The memory blocks are stored in order in each row, as shown in Figure 46(a). A path access reads and writes only a selected subset of data blocks in each row. Figure 46(b) compares the access probabilities of different blocks. While the root block (of the subtree) is always accessed, only one of its immediately two children would be accessed, resulting in 1/2 access probability for each node. The access probability decreases logarithmically as level increases.

When the top tree blocks are always mapped at the beginning of a page, the corresponding device locations tend to wear out much faster than other locations. We will develop intra-row wear leveling to exploit the unique ORAM access pattern. Each subtree (page) will maintain a shift vector which contains $log(Subtree_N)$ bits. It indicates the actual position of B_0 in the page. As each subtree left one empty cacheline in the page, the shift vector can be stored in the empty slots of the line. Every time the subtree is read to the upper ORAM controller, the shift vector will add one, and during the write phase, each block in the page will shift right to enable wear-leveling. For data blocks at different levels, we will evaluate if traditional wear leveling schemes [39, 58, 57] are still effective and develop new ones if necessary.

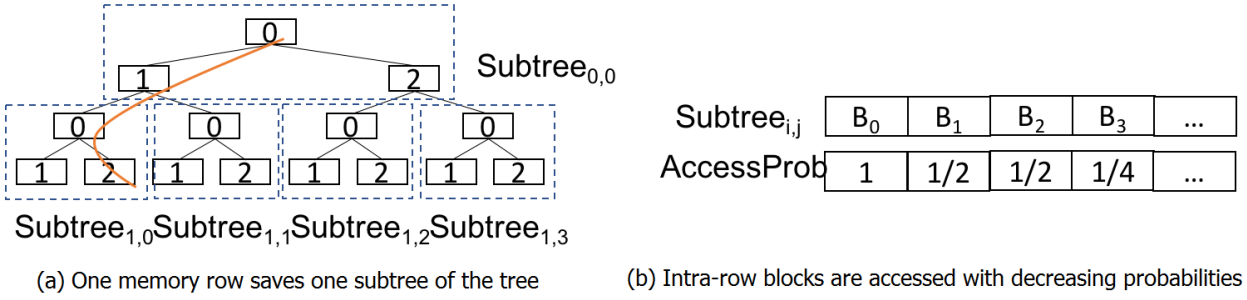


Figure 46: The ORAM Subtree layout and its block access probability.

6.2.2 Tradeoffs between retention time and write performance.

Given a Path ORAM tree, the tree nodes being close to the root are more likely to be accessed. Figure 47 depicts the access frequency decreases quickly as the level increases. This motivates the design of NVM write strategies with tradeoffs between retention time and write speed.

While NVM cells hold data after programming, their retention time, i.e., the duration that the data persist in the cells, is often a function of the write voltage and the magnitude of the write pulse. In the following discussion, we choose PCM as an example and define *weak write operation* as the write with low voltage and fast write speed [94]. Other memory

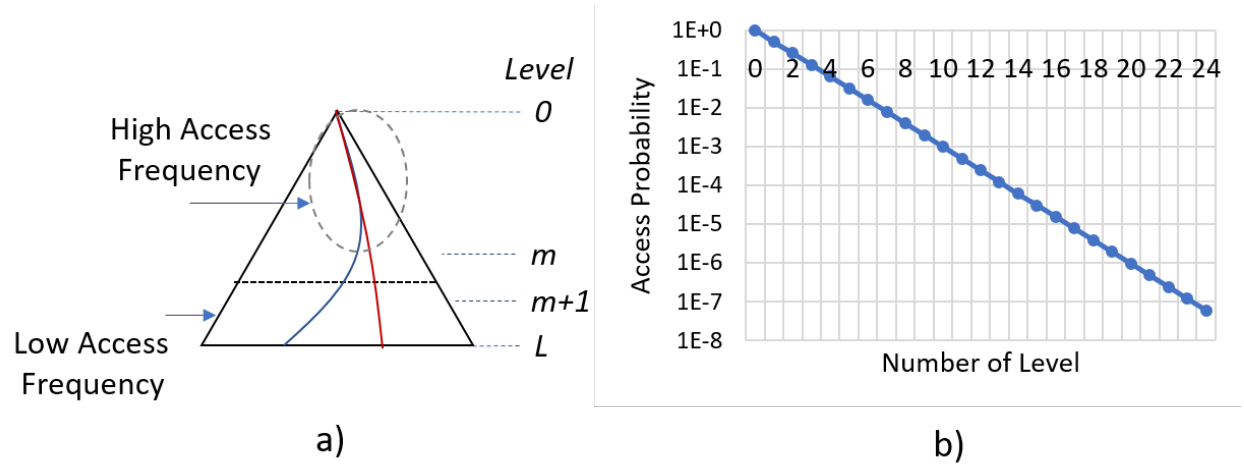


Figure 47: Access frequency of Path ORAM. a) The higher levels have higher access frequency. b) Theoretical access probability if we assume all path accesses are random.

technologies such as Flash, MLC STT-MRAM[48, 75] and ReRAM [49, 93] show similar write modes. They differ in voltage levels and write pulse widths, etc.

NVM cells need to be periodically refreshed (though at a much low frequency than that of DRAM) to ensure data integrity. Intuitively, writing NVM cells with weak write strategies leads to more frequent refresh operations. However, such refresh operations may be skipped if the data were just written. By exploiting the memory access frequency of tree nodes at different levels, we are to develop retention aware write modes for Path ORAM. In particular, we adopt weak write modes for nodes that are close to the root as such nodes may be accessed shortly in the future.

Such design has two benefits: 1) Previous write path operation can be finished earlier, 2) Allow next path access which has overlap with the previous access start earlier. Note: (1) while on-chip ORAM controller often adopts a large tree-top cache to buffer frequently accessed tree nodes, offloaded ORAM controllers often have limited buffer space. Depending on the integration of ORAM controllers, tree-top nodes may be accessed from NVM space and thus adopting fast write modes can effectively reduce the overall ORAM performance. (2) A secure application may have phases that have different memory access frequency,

i.e., MPKI (memory accesses per kilo instructions). To prevent information leakage, recent studies obfuscate program phases by generating dummy memory accesses. As a result, the next time that a tree node may be accessed can be computed with high accuracy.

6.3 SCALABLE ORAM DESIGN TO ALLOW MULTI-CLIENT ACCESSES

Given security protection has arisen as one of the top concerns in cloud computing, we envision that multiple applications running at the server side may demand strong privacy protection. We next develop scalable ORAM implementation to support concurrent Path ORAM accesses from multiple threads or processes. Our design will be built upon secure ORAM controller that we develop in preceding tasks. That is, the ORAM controller cooperates with the trusted processor for providing transparent security protection, in particular, privacy protection. The secret keys used to encrypt and obfuscate data in the ORAM tree are hidden from the data owners and service clients. By making Path ORAM a secure service, path obfuscation can finish within the ORAM controller. In addition, even when sensitive data are shared by multiple users, one user may not reveal the access patterns of others, even with the collusion from honest-but-curious servers.

6.3.1 Concurrent Path ORAM accesses to non-shared data.

We start with developing techniques to support concurrent Path ORAM accesses while there are no share data. The design challenge that we face in this scenario is the competition for ORAM controller hardware and DRAM space allocation.

Dynamic resource allocation between multiple ORAM applications. We propose to study and characterize the secure applications that demand ORAM protection and develop dynamic resource allocation strategies to guide the allocation of ORAM controller usage and DRAM space allocation. When adopting Path ORAM, a secure application generates ORAM accesses at a fixed rate to prevent timing attacks [24, 7]. It generates more dummy accesses if having a smaller MPKI and fewer dummy accesses if having a larger MPKI. Reducing the

rate at which the ORAM accesses (dummy or real) are generated would waste less memory bandwidth and have lower interference. However, such a strategy correlates the ORAM rate and the application's MPKI, which leaks information. We will study the cases that may tolerate such a leakage, for example, the application is well-known and we use its MPKI profiled with benchmark input, i.e., the leakage information is not bound to the user's input. In the project, we will study more allocation strategies and the tradeoff between security and performance.

To enable dynamic DRAM allocation, we may face additional design challenges. For example, ORAM space may become non-contiguous, which demands non-trivial path id to physical address mapping. Accessing an L-level ORAM tree path could demand up to L mapping, which can overburden TLB and degrade system performance considerably. We will study this and similar design challenges arising from dynamic resource allocation, and develop simple yet effective approaches to tradeoff among multiple designs factors.

6.3.2 Concurrent Path ORAM accesses to shared data.

For a more generalized application scenario, multiple applications may share some sensitive data while each application has its private data. For example, in a medical record database, patients' medical record may be accessible to both a doctor, a nurse, and an insurance company. While the doctor can change the record, the nurse and the insurance company may only read the record. In addition to enforcing the pre-defined access privileges, the nurse should not be able to tell if the insurance company has accessed some patients' records. In this project, we are to develop low overhead techniques to enforce these accesses.

Enhance Path ORAM with access control. We will study techniques to enhance Path ORAM with access control inside the ORAM controller. While access control has been widely adopted in OS to enable and adjust the accesses of shared data, our design takes advantage of *secure ORAM control*, which exhibits new design opportunities. In particular, for traditional access control, OS acts as a centralized controller, which greatly simplifies the control assignment and dynamic adjustment. To enforce privacy protection for all parties, adjust access control may demand dynamic negotiation from multiple participants.

We use an access control matrix with rows being different data chunks, columns being different applications; an entry being 1 or 0 indicate the corresponding application may and may not access the data chunk, respectively. At runtime, each ORAM access first checks the access control matrix and gets the access validated before accessing the real data. By adopting the *secure ORAM control* model, we expect minimized modification to existing Path ORAM implementation and small performance overhead at runtime. As a comparison, we eliminate the significant re-organization of Path ORAM tree, the public key encryption, and large data movement overheads in traditional design, e.g., GoRAM [51].

Figure 48 illustrates a new privacy attack that may arise in Path ORAM enhanced cloud servers. In the figure, we assume that two applications A and B can access shared data block b. Assume application A accesses b first and places it in the shared last level cache. When application B tries to access b, it returns within a short period of time (i.e., L2 access latency). However, if application B has never accessed b before. This short access latency would reveal application A's access behavior. This attack could become more serious if applications A and B build a covert channel through the last level cache. That is, depending on whether bit information '1' or '0' to send from application A to application B, application A may choose to access or do not access data block b after flushing the last level cache. Application B the receives the information by checking its access latency of data block b.

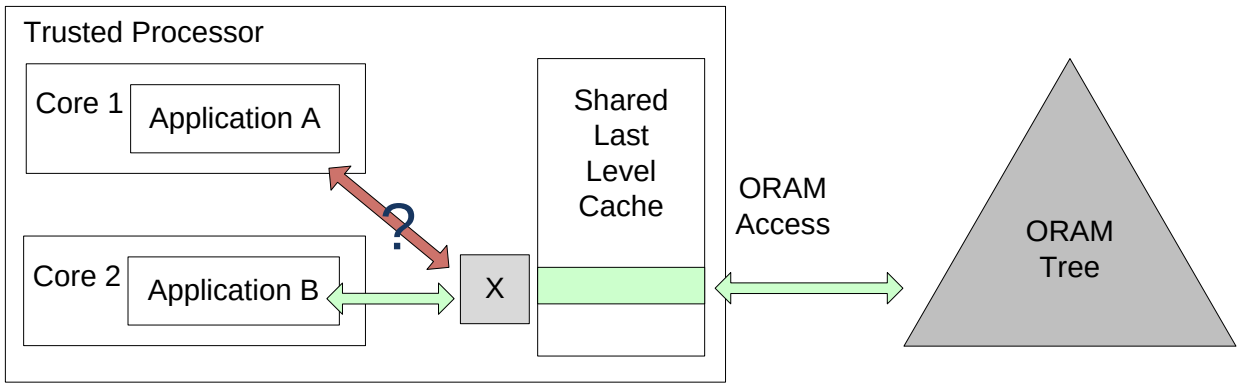


Figure 48: Potential covert channel due to shared ORAM and oblivious cache. X denotes the access control component to be developed in the project.

Oblivious shared cache. To protect the potential inter-client covert channel, we plan to design an oblivious shared cache, used together with ORAM. For every shared data entry in the cache read from ORAM, we need to set an additional access control flag in the cache. Only the client that originally fetched the data can read it from the cache. Other clients, which may have the access to the data entry, can only read it from ORAM even the data is present in the cache. As shown in Figure 48, user B will not be allowed to read or write shared entry x even it exists in the cache but was brought up by other users. User B will need another load from ORAM access to read the entry x in this case. With the oblivious shared cache design, plus ORAM, no inter-client access pattern will be leaked.

7.0 CONCLUSIONS

In this dissertation, I discussed and introduced several architectural solutions to build efficient secure memory system using ORAM, which can protect the memory system from access pattern leakage. The cost of ORAM algorithms comes from frequently shuffle and rewrite the memory content, which will saturate the system memory bandwidth and also compete with other co-run applications. Without effective scheduling technique or specialized design architecture, the ORAM algorithm and protected application can lead to significant co-run interference in the system. Therefore, there is a strong need to analyze ORAM accesses in architecture view and redesign the memory interface to enhance the application performance. This is where this dissertation lies in this field, by utilizing architectural enhancement techniques, our work makes ORAM even more practical to be adopted on a shared server environment.

My first work in Chapter 3, Cooperative Path ORAM design[80], is an optimized architecture design for memory bandwidth sharing between secure and non-secure applications. I propose practical buffer design to accelerate both types of applications without harming ORAM security invariants. The memory channel utilization is improved, and the CP-ORAM design achieves an average of 20% performance improvement over the baseline Path ORAM design while providing a flexible resource tuning between different kinds of applications.

The second work in Chapter 4, ORAM delegator[81], targets at the larger interference between different applications on the server when we scale the number of applications on the same physical machine. The major design upgrades memory channels by forming a secure engine on the buffer on board unit and offload the ORAM operations on the buffer. This design allows physical isolation of applications with limited pin counts out processor side while providing high parallelism to both types of applications.

The major differences of this dissertation from other work in this field are, we are the first to study the co-run interference on a shared server environment. Prior art, which focused on improving ORAM algorithms or ORAM access overhead, does not consider the situation that the physical machines may be shared by multiple different applications, therefore, the bandwidth is shared. Our work considered a more practical use scenario and studied the root cause of application slowdown from memory architecture view. Further, to our knowledge, our BoB based ORAM delegator design is the first design to provide isolated ORAM channel with high expansion capability and high internal parallelism processing capability, without sacrificing the security guarantee of the system. The major contributions of the thesis are summarized below:

- To our best of knowledge, this is the first work to study the ORAM interference with other applications on a shared server environment.
- First work to propose micro-architecture enhancement, such as DRAM command-level scheduling techniques, to accelerate applications and bandwidth utilization.
- First work to enable efficient and secure memory space expansion for ORAM based applications.
- Propose a secure memory architecture with no DRAM side interface or hardware modification, which is low cost compare to other state-of-art proposals.
- Enable a flexible priority tuning between different applications and present the design space exploration when designing scalable and secure memory architectures.

At the end of this dissertation, I discussed several other ongoing projects related to other architectural enhancements on ORAM based system, which aims to build ORAM based memory system more scalable regarding lower access latency, higher density, higher throughput, and concurrency. With each technique, the high overhead of ORAM can be reduced step by step, which enable this strong access pattern protection scheme more acceptable on a real system.

BIBLIOGRAPHY

[1] Shaizeen Aga and Satish Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 94–106. ACM, 2017.

[2] Arvind Arasu and Raghav Kaushik. Oblivious query processing. *arXiv preprint arXiv:1312.4012*, 2013.

[3] JEDEC Solid State Technology Association et al. Jedec standard: Ddr4 sdram. *JESD79-4, Sep*, 2012.

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

[5] Rajeev Balasubramonian. Making the case for feature-rich memory systems: The march toward specialized systems. *IEEE Solid-State Circuits Magazine*, 8(2):57–65, 2016.

[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4):36–42, 2014.

[7] Chongxi Bao and Ankur Srivastava. Exploring timing side-channel attacks on path-orams. In *Hardware Oriented Security and Trust (HOST), 2017 IEEE International Symposium on*, pages 68–73. IEEE, 2017.

[8] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, S Pugsley, A Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep*, 2012.

[9] Jie Chen and Guru Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 216–228. IEEE, 2014.

[10] Licheng Chen, Tianyue Lu, Yanan Wang, Mingyu Chen, Yuan Ruan, Zehan Cui, Yongbing Huang, Mingyang Chen, Jiutian Zhang, and Yungang Bao. Mims: Towards a message interface based memory system. *arXiv preprint arXiv:1301.0051*, 2013.

[11] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *2010 IEEE Symposium on Security and Privacy*, pages 191–206. IEEE, 2010.

[12] Hybrid Memory Cube Consortium et al. Hybrid memory cube specification rev. 2.0 (2013).

[13] Hybrid Memory Cube Consortium et al. Hybrid memory cube specification version 1.0. Technical report, Technical Report, http://www. hybridmemorycube. org/specification-download, 2013.

[14] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. *Buffer-on-board memory systems*. IEEE Computer Society, 2012.

[15] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[16] Zehan Cui, Tianyue Lu, Sally A McKee, Mingyu Chen, Haiyang Pan, and Yuan Ruan. Twin-load: Bridging the gap between conventional direct-attached and buffer-on-board memory systems. In *Proceedings of the Second International Symposium on Memory Systems*, pages 164–176. ACM, 2016.

[17] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Prefetch-guard: Leveraging hardware prefetchers to defend against cache timing channels. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2018.

[18] Kun Fang, Long Chen, Zhao Zhang, and Zhichun Zhu. Memory architecture for integrating emerging memory technologies. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 403–412. IEEE, 2011.

[19] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 283–295. IEEE, 2015.

[20] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C Myers, and G Edward Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 382–393. IEEE, 2016.

[21] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8. ACM, 2012.

[22] Christopher W Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket oram: Single online roundtrip, constant bandwidth oblivious ram. *IACR Cryptology ePrint Archive*, 2015:1065, 2015.

[23] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive oram:[nearly] free recursion and integrity verification for position-based oblivious ram. In *ACM SIGPLAN Notices*, pages 103–116. ACM, 2015.

[24] Christopher W Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 213–224. IEEE, 2014.

[25] Fujitsu. Sparc64 xifx:: Fujitsu's next generation processor for hpc. www.fujitsu.com, 2014.

[26] Brinda Ganesh, Aamer Jaleel, David Wang, and Bruce Jacob. Fully-buffered dimm memory architectures: Understanding mechanisms, overheads and scaling. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 109–120. IEEE, 2007.

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

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

[29] Akhila Gundu, Ali Shafiee Ardestani, Manjunath Shevgoor, and Rajeev Balasubramonian. A case for near data security. In *Workshop on Near-Data Processing*, 2014.

[30] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G Friedman. Ac-dimm: associative computing with stt-mram. In *ACM SIGARCH Computer Architecture News*, pages 189–200. ACM, 2013.

[31] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[32] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. Understanding contention-based channels and using them for defense. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 639–650. IEEE, 2015.

[33] Oracle Inc. Oracle sparc t7 and sparc m7 server architecture. Technical report, Oracle White Paper 2702877, 2016.

[34] Intel. Intel xeon processor e7 family. Technical report, https://www.intel.com/Intel Processors/Intel Xeon Processors, 2017.

[35] ISO. Iso/iec 11889-1:2009. Technical report, International Organization for Standardization, 2013.

[36] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[37] Joe Jeddeloh and Brent Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[38] JEDEC. Fbdimm: Architecture and protocol. JESD206.

[39] Lei Jiang, Yu Du, Youtao Zhang, Bruce R Childers, and Jun Yang. Lls: Cooperative integration of wear-leveling and salvaging for pcm main memory. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 221–232. IEEE, 2011.

[40] 2012 memory scheduling championship (msc). http://www.cs.utah.edu/~rajeev/jwac12/. Accessed: 2018-05-01.

[41] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. 2016.

[42] Mark LaPedus. Micron rolls ddr3 lrdimm. *EE Times*, 2009.

[43] TeleDyne Lecroy. Kibra 480 analyzer. http://teledynelecroy.com, Retrieved in, 2017.

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

[45] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418. IEEE, 2016.

[46] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[47] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention

time profiling mechanisms. In *ACM SIGARCH Computer Architecture News*. ACM, 2013.

[48] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing nand flash-based ssds via retention relaxation. *Target*, 11(10):00, 2012.

[49] Yongpan Liu, Zhibo Wang, Albert Lee, Fang Su, Chieh-Pu Lo, Zhe Yuan, Chien-Chen Lin, Qi Wei, Yu Wang, Ya-Chin King, et al. 4.7 a 65nm reram-enabled nonvolatile processor with 6× reduction in restore time and 4× higher clock frequency using adaptive data retention and self-write-termination nonvolatile logic. In *Solid-State Circuits Conference (ISSCC), 2016 IEEE International*, pages 84–86. IEEE, 2016.

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

[51] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 341–358. IEEE, 2015.

[52] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[53] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160. IEEE Computer Society, 2007.

[54] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[55] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers??? Track at the RSA Conference*, pages 1–20. Springer, 2006.

[56] J. T. Pawlowski. Hybrid memory cube (HMC). In *Hot Chips*, 2011.

[57] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 14–23. IEEE, 2009.

[58] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[59] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. *IACR Cryptology ePrint Archive*, 2014:997, 2014.

[60] Ling Ren, Christopher W Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Design and implementation of the ascend secure processor. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[61] Ling Ren, Xiangyao Yu, Christopher W Fletcher, Marten Van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious ram in secure processors. In *ACM SIGARCH Computer Architecture News*, pages 571–582. ACM, 2013.

[62] Scott Rixner. Memory controller optimizations for web servers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 355–366. IEEE Computer Society, 2004.

[63] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. Memory access scheduling. In *ACM SIGARCH Computer Architecture News*, pages 128–138. ACM, 2000.

[64] Gururaj Saileshwar, Prashant J Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K Qureshi. Synergy: Rethinking secure-memory design for error-correcting memories. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 454–465. IEEE, 2018.

[65] Ali Shafiee, Rajeev Balasubramonian, Mohit Tiwari, and Feifei Li. Secure dimm: Moving oram primitives closer to memory. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, pages 428–440. IEEE, 2018.

[66] Patrick Simmons. Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 73–82. ACM, 2011.

[67] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, et al. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1), 2015.

[68] JEDEC Standard. Ddr3 sdram standard. *JESD79-3, Jun*, 2007.

[69] JEDEC Standard. High bandwidth memory (hbm) dram. *JESD235*, 2013.

[70] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 253–267. IEEE, 2013.

[71] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

[72] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol.

In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310. ACM, 2013.

[73] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 339. IEEE Computer Society, 2003.

[74] G Edward Suh, Charles W O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ACM SIGARCH Computer Architecture News*, pages 25–36. IEEE Computer Society, 2005.

[75] Zhenyu Sun, Xiuyuan Bi, Hai Helen Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. Multi retention level stt-ram cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 329–338. ACM, 2011.

[76] Nexus Technology. Ma5100/4100 series memory analyzer.

[77] Tektronix. Memory interface electrical verification and debug ddr. http://www.tek.com, Retrieved in, 2017.

[78] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 177–188. IEEE Press, 2013.

[79] Hao Wang, Chang-Jae Park, Gyung-su Byun, Jung Ho Ahn, and Nam Sung Kim. Alloy: Parallel-serial memory channel architecture for single-chip heterogeneous processor systems. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 296–308. IEEE, 2015.

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

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

[82] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 225–236. IEEE, 2014.

[83] Yao Wang, Benjamin Wu, and G Edward Suh. Secure dynamic memory scheduling against timing channel attacks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 301–312. IEEE, 2017.

[84] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: A parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 977–988. ACM, 2012.

[85] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 347–360. ACM, 2017.

[86] Jun Yang, Youtao Zhang, and Lan Gao. Fast secure processor for inhibiting software piracy and tampering. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 351. IEEE Computer Society, 2003.

[87] Fan Yao, Miloš Doroslovački, and Guru Venkataramani. Are Coherence Protocol States Vulnerable to Information Leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[88] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of ACM Great Lakes Symposium on VLSI (GLSVLSI)*, pages 155–160. ACM, 2017.

[89] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, pages 719–732, 2014.

[90] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 313–324. IEEE, 2017.

[91] Doe Hyun Yoon, Jichuan Chang, Naveen Muralimanohar, and Parthasarathy Ranganathan. Boom: Enabling mobile memory based low-power server dimms. In *ACM SIGARCH Computer Architecture News*, pages 25–36. IEEE Computer Society, 2012.

[92] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Proram: dynamic prefetcher for oblivious ram. In *ACM SIGARCH Computer Architecture News*, pages 616–628. ACM, 2015.

[93] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T Chong. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 519–531. IEEE, 2016.

[94] Mingzhe Zhang, Lunkai Zhang, Lei Jiang, Zhiyong Liu, and Frederic T Chong. Balancing performance and lifetime of mlc pcm by using a region retention monitor. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 385–396. IEEE, 2017.

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

[96] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. In *ACM SIGPLAN Notices*, pages 72–84. ACM, 2004.