2018 IEEE International Symposium on High Performance Computer Architecture

# D-ORAM: Path-ORAM Delegation for Low Execution Interference on Cloud Servers with Untrusted Memory

Rujia Wang† Youtao Zhang§ Jun Yang†
† *Electrical and Computer Engineering Department* § *Computer Science Department*
*University of Pittsburgh*
{*ruw16,youtao,juy9*}*@pitt.edu*

***Abstract*—Cloud computing has evolved into a promising computing paradigm. However, it remains a challenging task to protect application privacy and, in particular, the memory access patterns, on cloud servers. The Path ORAM protocol achieves high-level privacy protection but requires large memory bandwidth, which introduces severe execution interference. The recently proposed secure memory model greatly reduces the security enhancement overhead but demands the secure integration of cryptographic logic and memory devices, a memory architecture that is yet to prevail in mainstream cloud servers.**

**In this paper, we propose D-ORAM, a novel Path ORAM scheme for achieving high-level privacy protection and low execution interference on cloud servers with untrusted memory. D-ORAM leverages the buffer-on-board (BOB) memory architecture to offload the Path ORAM primitives to a secure engine in the BOB unit, which greatly alleviates the contention for the off-chip memory bus between secure and non-secure applications. D-ORAM upgrades only one secure memory channel and employs Path ORAM tree split to extend the secure application flexibly across multiple channels, in particular, the non-secure channels. D-ORAM optimizes the link utilization to 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.**

## I. Introduction

Cloud computing has evolved as a ubiquitous computing paradigm nowadays. To maximize hardware resource utilization and reduce energy consumption, cloud providers widely adopt server consolidation to share the same hardware resources among multiple co-running applications. However, such an execution model raises security concerns. On the one hand, a curious or malicious server may monitor the execution, e.g., it may attach physical devices to eavesdrop the memory communication [26], [37]. On the other hand, a co-running application may extract sensitive information through covert communication channels [42], [47].

To ensure high-level security protection, the processor chip needs to integrate security engines to defend against various attacks. Intel Software Guard Extensions (SGX) [20] isolates the code and data of private enclave functions from the rest of system. By including the processor chip as the only hardware component in the trusted computing base (TCB), the XOM (execution only memory) model saves encrypted data and code in the untrusted memory [27], [35], which effectively protects data confidentiality. Recent studies revealed that protecting data privacy on untrusted memory demands oblivious memory (ORAM) to reshuffle memory data after each memory access [16]. Unfortunately, ORAM often introduces large memory contention and performance degradation. For example, in the recently proposed Path ORAM scheme [34], one memory access from the application is converted to tens to hundreds of memory accesses, exhibiting extreme memory access intensity [34].

To alleviate the performance overhead introduced in Path ORAM, the secure memory based designs, e.g., ObfusMem [3] and InvisiMem [2], place both the processor chip and the main memory module in the TCB. The secure memory model protects data privacy through communication channel encryption, which has low-performance overhead in general. However, it requires the secure integration of cryptographic logic and memory devices. For example, adding a secure (bridge) chip to the DRAM DIMM cannot meet the model requirement as the wires on the PCB (printed circuit board) may be compromised for eavesdropping. Placing the secure engine in the logical layer of HMC (hyper memory cube) architecture is viable [2] as the connection between logic and memory devices are embedded inside one package. However, HMC faces fabrication challenges on module capacity and yield. The mainstream computing servers still widely adopt traditional untrusted DRAM modules. To summarize, it is important to devise low interference privacy protection schemes for cloud servers with untrusted memory.

In this paper, we propose D-ORAM, a novel oblivious memory scheme, for cloud servers with untrusted memory. D-ORAM achieves the good tradeoff among high-level security protection, low execution interference, and good compatibility with existing server architecture. The following lists our contributions.

- We propose to have Path ORAM delegated in a small off-chip secure engine. D-ORAM leverages the BOB (buffer-on-board) architecture such that the TCB consists of the processor and the small secure delegator embedded in the BOB unit. The secure delegator offloads the expensive

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00043

Path ORAM primitives from the processor, which effectively mitigates the extreme memory contention at the on-chip memory controller. The precious processor resources can be better exploited by co-running applications.

- We analyze the channel utilization and space allocation in the new memory architecture. We split the Path ORAM tree across the secure memory channel and other non-secure ones, which achieves flexible space allocation without adding more components to the TCB. We propose utilization aware data allocation of non-secure application, eliminating the potential bottleneck of the secure channel.
- We evaluate the proposed scheme and compare it to the state-of-the-art. Our evaluation shows that D-ORAM integrates well with server memory architectures and effectively achieves low co-run interference. It improves the NS-App performance by 22.5% on average over the Path ORAM baseline.

In the rest of the paper, we briefly introduce the background and motivate the design in Section II. Section III elaborates the proposed D-ORAM. We present the experimental methodology and analyze the results in Section IV and Section V, receptively. Additional related work is discussed in Section VI, We conclude the paper in Section VII.

## II. Background and Motivation

In this section, we briefly compare different DRAM memory architectures and then discuss the threat model and the privacy protection in two existing secure execution models.

### A. The Memory System Architecture

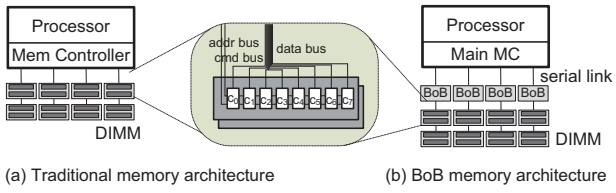


Figure 1. The DRAM based memory system architectures.

The DRAM based memory system traditionally adopts the direct-attached memory architecture, as shown in Figure 1(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 [24]. Different memory channels may be ganged together, i.e., operate synchronously, to form wide data buses.

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 redrives the signal to boost signal integrity [25], to a buffer-on-board (BOB) unit that controls the DRAM and receives requests and sends data back to the processor [9], to the HMC architecture [29] 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 [26], [37].

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 [33]. Oracle M7 supports up to 16 DIMMs using eight BOB buffers and 1TB per processor [19]. Intel Xeon E7 [21] 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 [23]. 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 [14], which was released in 2015 and connects to 32GB memory using eight 4GB HMC modules.

### B. The TCB and The 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 [22]. 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 [32], [13], [44], 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 paper, the processor chip is included in the TCB, similar to previous designs [27], [32], [2], [3]. 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.

*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 2(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 [27]. Suh *et al.* proposed Merkle tree based verification to efficiently check the integrity of memory that contains dynamic data [36].

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 [8]. 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 [45].

**ORAM Model.** Studies have shown that to securely prevent information leakage from memory access patterns, it demands oblivious memory (ORAM) primitives [16], [17]. 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 [34] was recently proposed as a practical ORAM implementation. Figure 3 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 [32].

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 $23\times4$ blocks if only the root node is cached, or $21\times4$ blocks if top 3 levels are cached, etc. These accessed blocks can be physically mapped to multiple memory channels to increase parallelism.

*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 2(b). The recent proposed ObfusMem [3] and InvisiMem [2] 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.

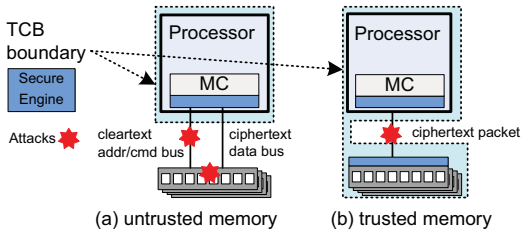


Figure 2. Comparing two TCB models.

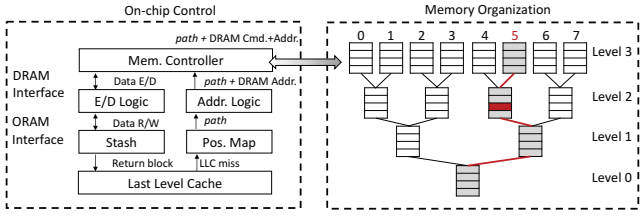


Figure 3. Basic Path ORAM Organization and Operations

### C. 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 4 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 [1]. 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).

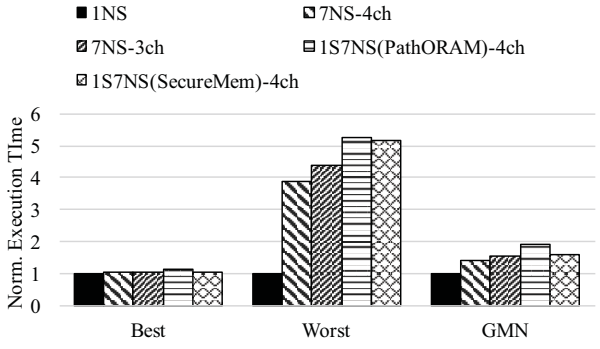


Figure 4. The performance degradation under different co-run scenarios.

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 [32], [39], 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 adopt 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. By giving one more channel resource, `7NS-4ch` shows 43% slowdown.

Adopting secure memory execution model is beneficial but not significant. We modeled the channel replication and read write obfuscation as in ObfusMem[3] and InvisiMem[2]. While the secure memory execution model introduces around 10% to S-App execution (as in [3]), 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.

### D. Design Goal

In this paper, 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.

## III. The D-ORAM Details

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

### A. An Overview

An overview of the D-ORAM memory system is illustrated in Figure 5. 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.

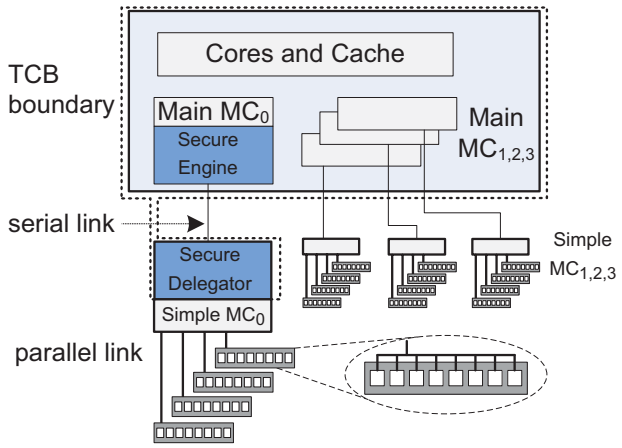


Figure 5. An overview of D-ORAM memory system.

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*.

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 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.

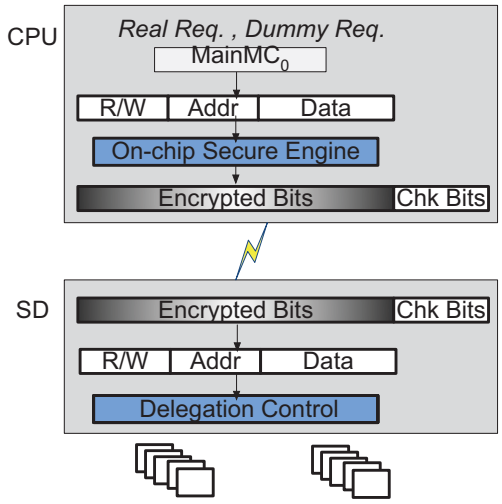


Figure 6. Delegating Path ORAM in SD.

### B. 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 6. 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.

(1) 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 [2], [3]. For the read requests, the data field contains dummy data, e.g., all 0s.

(2) D-ORAM, in addition to a real memory request from the secure application, may generate dummy requests to prevent timing channel attack [28], [46].
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 paper. If there is no real request from S-App, a dummy request is generated and sent.

(3) D-ORAM adopts the 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 [2].
The on-chip secure engine generates a 72B-long $OTP$ using AES encryption, and XOR the $OTP$ and the packet as follows.

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

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.

(4) 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 [3].

**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 3). We will evaluate its hardware overhead in Section III-E.

The processing follows the Path ORAM protocol [34]. 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.

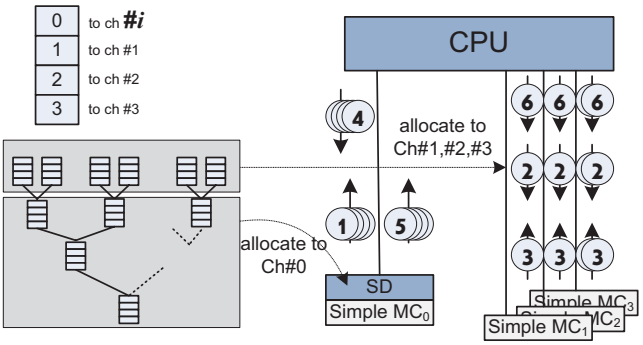


Figure 7. Splitting a Path D-ORAM tree across channels.

**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 paper, we set $t$=50. If the write phase is ongoing when the processor chip receives the response packet, we choose to buffer the new request in SD and get it serviced after the write phase for the current request.

Given that sub-channels use parallel links, SD sends the request to the simple controller, which is responsible for generating the detailed device commands and device access time to the insecure memory.

### *C. 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 [32], [39], 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% [34]. 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.

Table I
BALANCE SPACE DEMAND ACROSS CHANNELS

| k | Data Block Distribution | | Extra Messages | |
|---|---|---|---|---|
| | channel #0 | channel #1,#2,#3 | channel #0 | channel #1,#2,#3 |
| 1 | 50.0% | 16.7% per channel | 4k **short** Read packets, 4k Response packets, 4k Write packets | $m$ **short** Read packets, $m$ Response packets, $m$ Write packets, $m \in$[k,2k] |
| 2 | 25.0% | 25.0% per channel | | |
| 3 | 12.5% | 29.2% per channel | | |

We then propose to balance the space demand by expanding the Path ORAM tree across channels. As shown in Figure 7, we observe that the nodes in the last level of the 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 7. Table I 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.

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 I 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.

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

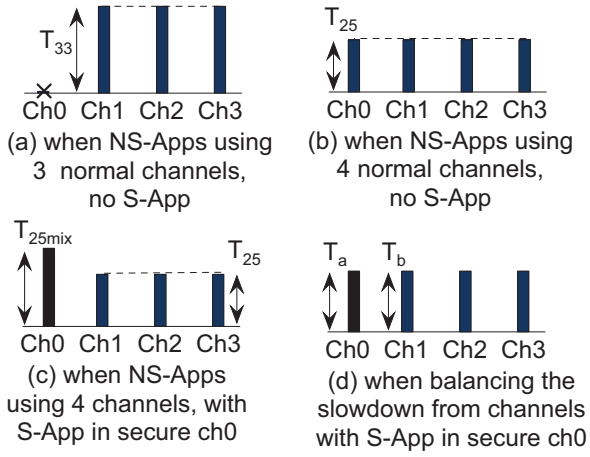


Figure 8. Balance the average access latency between secure and normal channels.

### D. 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 [39], 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 8 illustrates the memory access latency when we allocate NS-Apps using different memory channels. Figure 8(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 8 (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. 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 8(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 V-C .

### E. 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 [31], 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.

### F. Extension to Parallel Link Buses

In this paper, 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 [11], 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.

### G. Security Analysis

Our D-ORAM design focuses on reducing the application execution interference by delegating the secure engine to BoB unit. The access pattern of S-App is still protected by Path ORAM which does not show any information leakage through the plaintext on the conventional directed attached interface. We do not change any protocol of Path ORAM, hence, the protection strength is not affected. In our co-run model, we assume that multiple applications sharing the same memory bandwidth. Our fix memory access rate for S-App can prevent timing side channel attack, as studied in previous work[44], [46].

## IV. 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[6]. 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 II summarizes the baseline processor and memory configuration. The DRAM memory follows JEDEC DDR3-1600 specification. We adopted the default values in the specification 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[32].

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 [39]. We set the threshold to 50% so that both kinds of applications have similar slowdown.

We choose 15 memory intensive benchmarks used in MSC [1]. 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 [1].

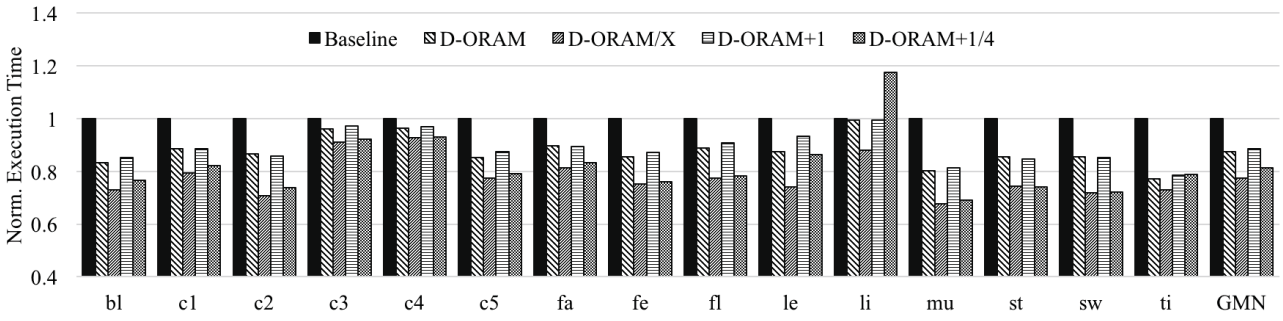


Figure 9. Comparing NS-App performance when adopting different D-ORAM schemes

Table II
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[10] |

Table III
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) |

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 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 III 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.

## V. RESULTS

In the experiments, we evaluated the following schemes:

- `Baseline`. This is the baseline for comparison purpose. It uses 4-channel direct-attached DRAM interface to run the workloads. The results of other schemes are normalized to `Baseline`.
- `D-ORAM`. This scheme implements 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.

### A. Performance Evaluation

We first analyzed the performance under different protection settings. Figure 9 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

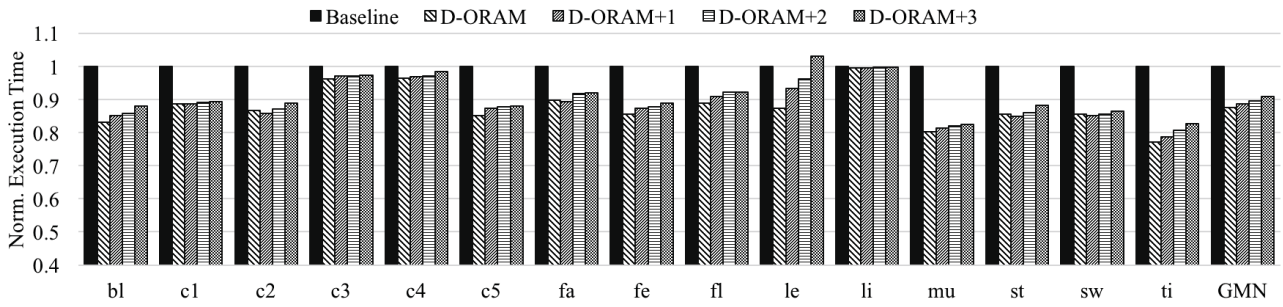


Figure 10. Comparing the performance impact when using large Path ORAM trees.

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.

### B. Expanding the Path ORAM Tree

Figure 10 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 $4\times2^k$GB.

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.

### C. Secure Channel Sharing

We then studied the effectiveness of secure channel sharing. Figure 11compares 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 12. 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 12, there is only one exception $c2$, 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.

### D. Access Latency Reduction

We also compared the average NS-App access latency reduction in Figure 13. In this experiment, for illustration purpose, we chose D−ORAM+1 and D−ORAM/4for 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.

### E. 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 [3], [32]. The overhead from BOB architecture is small.

## VI. Related Work

**ORAM Optimization.** The large performance overhead of ORAM has been a focus of recent ORAM designs. Ring ORAM[30], Bucket ORAM[12] were proposed to reduce the bandwidth overhead on the memory bus by using different bucket organization and more complicated access flow control.

To improve Path ORAM performance on DRAM based system, several techniques have been proposed. Ren et al.[32] 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. [44] eliminated unnecessary memory accesses if consecutive path accesses

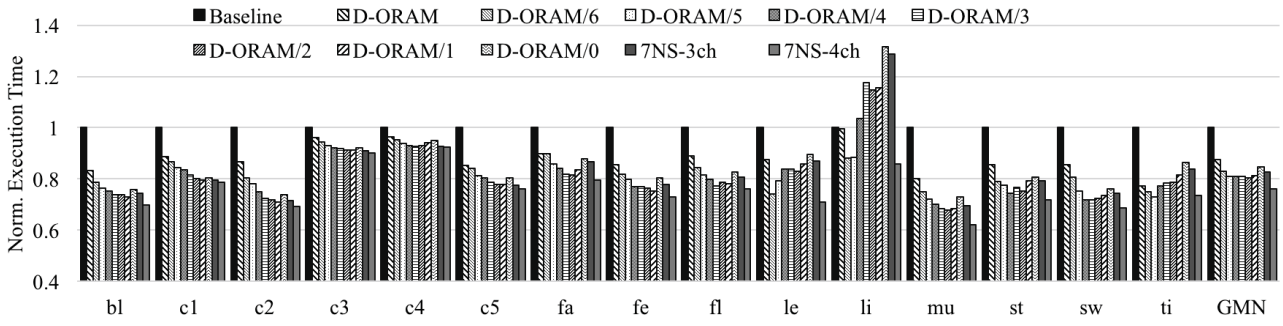


Figure 11. The performance impact when adopting secure channel sharing.

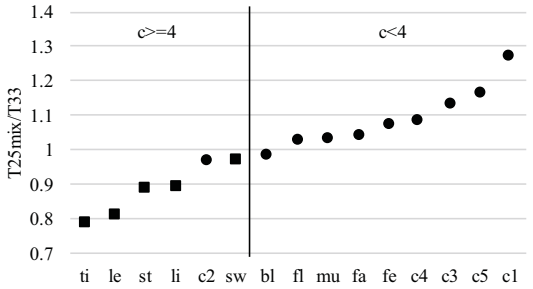


Figure 12. The ratio of $T_{25mix}$ and $T_{33}$. (Actual optimal result from Figure 11 is categorized as: •: best configuration $c < 4$, ■: best configuration $c >= 4$.)

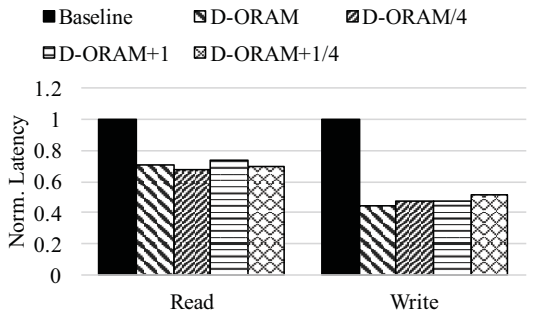


Figure 13. Comparing NS-App memory access latency.

have overlaps. Wang et al.[39] proposed an efficient bandwidth sharing technique, and read and write phase acceleration for ORAM applications co-run with other applications on server with conventional memory interface.

**Secure Memory.** The recent proposed ObfusMem [3] and InvisiMem [2] schemes assume that memory is secure, which can be exploited to significantly reduce protection overhead. Motivated by near data processing [4], [5], Gundu et.al propose to use a secure DIMM [18] 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.

To protect timing channel leakage, Wang *et al.* [40] 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[41].

**Covert channel attacks.** [47] propose 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. [48] use information theory to quantify the communication capacity of microarchitectural covert channels, and introduce a detection technique for covert channel eavesdropping attacks.

**DRAM Architectures.** Recent studies proposed alternative memory architectures to alleviate the constraints in traditional memory systems. Fully Buffered DIMM [15] adds a buffer on DIMM to handles memory requests. While FB-DIMM can effectively increase memory density, it introduces high power consumption, making it a less popular architecture in modern servers. BOOM [43] 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[7] 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[38] utilizes DDR based parallel interface as well as serial interface for a GPU enabled heterogeneous system.

## VII. Conclusions

In this paper, 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.

## ACKNOWLEDGMENT

We thank all the anonymous reviewers for their valuable comments and suggestions. This work is supported in part by US CCF 1718080 and CCF 1617071.

## REFERENCES

[1] "2012 Memory Scheduling Championship (MSC)," http://www.cs.utah.edu/~rajeev/jwac12/, accessed: 2017-07-01.
[2] S. Aga and S. Narayanasamy, "InvisiMem: Smart Memory Defenses For Memory Bus Side Channel," *ISCA*, 2017.
[3] A. Awad, Y. Wang, *et al.*, "ObfusMem: A Low-Overhead Access Obfuscation For Trusted Memories," *ISCA*, 2017.
[4] R. Balasubramonian, "Making the Case for Feature-Rich Memory Systems: The March Toward Specialized Systems," *IEEE Solid-State Circuits Magazine*, 2016.
[5] R. Balasubramonian, J. Chang, *et al.*, "Near-data processing: Insights from a MICRO-46 workshop," *IEEE Micro*, 2014.
[6] N. Chatterjee, R. Balasubramonian, *et al.*, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.
[7] L. Chen, et al., "Mims: Towards a message interface based memory system," *arXiv preprint arXiv:1301.0051*, 2013.
[8] S. Chen, R. Wang, *et al.*, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," *IEEE S&P*, 2010.
[9] E. Cooper-Balis, P. Rosenfeld, *et al.*, "Buffer-on-board Memory Systems," *ISCA*, 2012.
[10] Z. Cui, T. Lu, *et al.*, "Twin-Load: Bridging the Gap between Conventional Direct-Attached and Buffer-on-Board Memory Systems," *MEMSYS*, 2016.
[11] K. Fang, L. Chen, *et al.*, "Memory Architecture for Integrating Emerging Memory Technologies," *PACT*, 2011.
[12] C. W. Fletcher, M. Naveed, *et al.*, "Bucket ORAM: Single Online Roundtrip, Constant Bandwidth Oblivious RAM." *IACR Cryptology ePrint Archive*, 2015.
[13] C. W. Fletcher, L. Ren, *et al.*, "Freecursive ORAM: [Nearly] Free Recursion And Integrity Verification For Position-based Oblivious RAM," *ASPLOS*, 2015.
[14] Fujitsu, "SPARC64 XIfx:: Fujitsu's Next Generation Processor for HPC," www.fujitsu.com, 2014.
[15] B. Ganesh, A. Jaleel, *et al.*, "Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling," *HPCA*, 2007.
[16] O. Goldreich, "Towards A Theory of Software Protection And Simulation By Oblivious RAMs," *STOC*, 1987.
[17] O. Goldreich and R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *Journal of ACM*, 43, 1996.
[18] A. Gundu, A. S. Ardestani, *et al.*, "A case for near data security," 2nd Workshop on Near-Data Processing, 2014.
[19] O. Inc., "Oracle SPARC T7 and SPARC M7 Server Architecture," *Technical report*, Oracle White Paper 2702877, 2016.
[20] Intel, "Intel Software Guard Extension," https://software.intel.com/en-us/sgx, 2013.
[21] Intel, "Intel Xeon Processor E7 Family," *Technical report*, https://www.intel.com/Intel Processors/Intel Xeon Processors, 2017.
[22] ISO, "ISO/IEC 11889-1:2009," *Technical report*, International Organization for Standardization, 2013.
[23] JEDEC, "FBDIMM: Architecture and Protocol," JESD206.
[24] JEDEC, "DDR4 SDRAM Specification," JEDEC standard, 2012.
[25] M. LaPedus, "Micron rolls DDR3 LRDIMM," *EE Times*, 2009.
[26] T. Lecroy, "Kibra 480 Analyzer," http://teledynelecroy.com, Retrieved in, 2017.
[27] D. Lie, C. Thekkath, *et al.*, "Architectural Support for Copy and Tamper Resistant Software," *ASPLOS*, 2000.
[28] M. Maas, E. Love, *et al.*, "PHANTOM: Practical Oblivious Computation in a Secure Processor," *CCS*, 2013.
[29] J. T. Pawlowski, "Hybrid memory cube (HMC)," *Hot Chips*, 2011.
[30] L. Ren, C. W. Fletcher, *et al.*, "Ring ORAM: Closing the Gap Between Small and Large Client Storage Oblivious RAM." *IACR Cryptology ePrint Archive*, 2014.
[31] L. Ren, C. W. Fletcher, *et al.*, "Design and Implementation of the Ascend Secure Processor," *IEEE TDSC*, 2017.
[32] L. Ren, X. Yu, *et al.*, "Design Space Exploration And Optimization Of Path Oblivious RAM In Secure Processors," *ISCA*, 2013.
[33] B. Sinharoy, J. A. V. Norstrand, *et al.*, "IBM POWER8 Processor Core Microarchitecture," *IBM Journal of Research and Development*, 59(1), 2015.
[34] E. Stefanov, M. van Dijk, *et al.*, "Path ORAM: An Extremely Simple Oblivious RAM Protocol," *CCS*, 2013.
[35] G. E. Suh, D. Clarke, *et al.*, "Efficient Memory Integrity Verification and Encryption for Secure Processors," *MICRO*, 2003.
[36] G. E. Suh, D. E. Clarke, *et al.*, "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing," *SC*, 2003.
[37] Tektronix, "Memory Interface Electrical Verification and Debug DDR," http://www.tek.com, Retrieved in, 2017.
[38] H. Wang, C.-J. Park,*et al.*, "Alloy: Parallel-serial memory channel architecture for single-chip heterogeneous processor systems," *HPCA*, 2015.
[39] R. Wang, Y. Zhang, and J. Yang, "Cooperative Path-ORAM For Effective Memory Bandwidth Sharing In Server Settings," *HPCA*, 2017.
[40] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller," *HPCA*, 2014.
[41] Y. Wang, B. Wu, and G. E. Suh, "Secure Dynamic Memory Scheduling against Timing Channel Attacks," *HPCA*, 2017.
[42] Z. Wang and R. B. Lee, "Covert and Side Channels Due to Processor Architecture," *ACSAC*, 2006.
[43] D. H. Yoon, J. Chang,*et al.*, "BOOM: Enabling mobile memory based low-power server DIMMs," *ACM SIGARCH Computer Architecture News*, 2012.
[44] X. Zhang, G. Sun, *et al.*, "Fork Path: Improving Efficiency Of ORAM By Removing Redundant Memory Accesses," *MICRO* 2015.
[45] X. Zhuang, T. Zhang, and S. Pande, "HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus," *ASPLOS*, 2004.
[46] C. Fletchery, L. Ren, *et al.*, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs",*HPCA*,2014
[47] J. Chen and G. Venkataramani, "Cc-hunter: Uncovering covert timing channels on shared processor hardware",*MICRO*,2014
[48] C. Hunger, M. Kazdagli, *et al.*, "Understanding contention-based channels and using them for defense",*HPCA*,2015