2018 IEEE 36th International Conference on Computer Design

Breaking the Oblivious-RAM Bandwidth Wall

Hamza Omar*, Syed Kamran Haider*, Ling Ren†, Marten van Dijk*, Omer Khan*
†Massachusetts Institute of Technology, Cambridge, MA, USA
*University of Connecticut, Storrs, CT, USA

***Abstract*—PathORAM is a popular security primitive for obfuscating memory access patterns from a secure processor to an insecure main memory. Emerging throughput multicore and GPU processors provide immense memory bandwidth via multiple on-chip memory controllers. PathORAM translates a single off-chip cache line access into ∼100 cache lines, thereby stressing the available memory bandwidth. However, current PathORAM scheme shows degradation of bandwidth utilization with an increase in the number of memory controllers. This deprivation in bandwidth utilization is primarily due to the fact that PathORAM falls short in proportionate distribution of memory accesses among all available on-chip memory controllers. This paper presents a novel ORAM path distribution scheme that ensures balanced load distribution among parallel on-chip memory controllers, and consequently improves secure processor performance by ∼24% over state-of-the-art PathORAM scheme.**

I. Introduction

Privacy of user's data in computation outsourcing is becoming increasingly important in the era of data-center computing. Various *trusted hardware* based secure processor architectures [1], [2] have been proposed to preserve privacy. They rely on the mechanism of data *encryption and decryption* to provide users with secure access to the results of their computation. However, data encryption alone cannot provide sufficient protection to users' data in outsourced storage applications in the presence of a minimal trusted computing base (TCB) [3]. Zhuang et al. in [4] and John et al. in [5] have demonstrated that information in the main memory can be leaked to an adversary via access patterns to the stored data, even if the data is encrypted. Various data *encryption–decryption* based schemes [6], [7] have been proposed that provide a secure and efficient solution to preserve user's data privacy. However, these schemes rely on expanding the TCB to include the main memory. To prevent information leakage via memory access patterns while keeping the TCB to the minimum, Oblivious RAM [8] algorithms have been proposed that allow a client to conceal its access pattern to the remote storage. Among various ORAM algorithms, PathORAM [9] is currently the most efficient and well studied [10], [11], [12] ORAM implementation for secure processor architectures. However, PathORAM still incurs high latency overheads due to its many fold increase in the demand to access the main memory.

With the advent of many-core parallel architectures, systems such as Intel® Xeon Phi™ [13], and GPUs [14] feature a large number of energy-efficient cores alongside an immense memory bandwidth [15]. Moreover, with the debut of High-Bandwidth Memory [16], [17] and Hybrid Memory Cube (HMC) [18], [19], the memory latency and bandwidth of such systems has been greatly improved. Such advanced memory architectures provide high memory bandwidth via multiple on-chip memory controllers – vault controllers in case of HMCs. Given the trends for the last two decades, memory bandwidth will likely continue to improve. Indeed, with the continuing progress in 3-D integration, system's memory bandwidth is more likely to reach the numbers of 640 to 1000GB/Sec (reported by NVidia [20]). In a nutshell, effortlessly providing systems with high memory bandwidth is becoming the next major step for commodity single-chip parallel architectures [20].

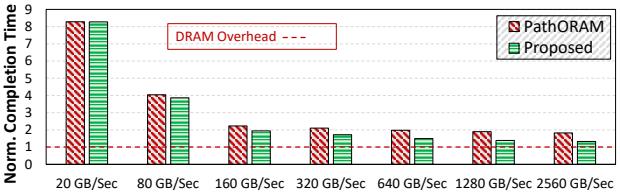

Fig. 1: *Reduction of ORAM performance overheads with the increase in memory bandwidth.*

High memory bandwidth can potentially result in reducing severe memory competition and performance overheads in PathORAM, as shown in Figure 1. We observe a significant reduction in PathORAM overheads from ∼8.3× (at 20GB/Sec total bandwidth) to ∼1.82×, when a total memory bandwidth of 2560GB/sec is provided. However, bandwidth utilization of the current PathORAM scheme does not scale with the increase in memory controllers [10], [11]. This is due to the fact that as memory controllers increase, the PathORAM mapping scheme falls short of distributing the requested path in an equally balanced manner. Therefore, it results in under-utilizing the available memory bandwidth. It is empirically observed that for a fixed total memory bandwidth, the utilization of PathORAM degrades from ∼96% to ∼76% as the on-chip memory controllers are increased from 2 to 32. PathORAM performance overheads can be reduced if all memory controllers utilize their peak bandwidth, as shown in Figure 1. However, ensuring peak bandwidth utilization in a multiple channel PathORAM setting requires proper balancing of off-chip memory accesses such that each controller contributes equally upon a path access. In addition to load balancing, the memory space division must also be balanced. Otherwise, memory controllers manage an unequal number of cache lines leading to under-utilization of resources. Therefore, the key challenge is *to gravitate PathORAM towards emerging systems with multiple memory controllers, while achieving near-peak memory bandwidth utilization*.

Achieving the aforementioned goal requires an intelligent mapping strategy to provide elegant load balancing among memory controllers. This paper proposes an ORAM path distribution scheme in which one ORAM controller monitors requests from the cores and directs them to corresponding DRAM controllers. Such monitoring allows balanced distribution of requests among the available memory controllers. The ORAM tree is segregated among sub-trees and each

2576-6996/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCD.2018.00026

path of the tree is split into data cache lines, such that each memory controller is assigned an equal number of blocks of the path – providing equal memory space division. In addition to this, the proposed scheme ensures proportionate cache line distribution among memory channels within the ORAM tree, thereby ensuring proper load distribution and improved bandwidth utilization. The improvement in bandwidth utilization provides higher gains with increasing total bandwidth. At memory bandwidth of $2560GB/Sec$, the proposed ORAM scheme is observed to perform competitively with the DRAM counterpart.

This paper introduces the first tree-based ORAM scheme for emerging high memory bandwidth architectures that utilize the peak memory bandwidth by assuring load and capacity balance among all memory controllers. Splash-2 and Database Management benchmarks achieve an average performance improvement of $\sim$22% over state-of-the-art PathORAM with multiple memory controllers. Moreover, machine learning and graph analytic workloads are analyzed to understand the impact of PathORAM on these emerging applications. The proposed scheme shows an average performance improvement of $\sim$29% for these workloads. It is worthwhile noting that these performance improvements are observed while keeping the TCB limited to the processor chip package. Overall, the average ORAM overheads are reduced from $\sim 2\times$ to $\sim 1.5\times$ through the proposed load balancing scheme at a memory bandwidth of $640GB/Sec$.

II. Secure Processor Background

A. Threat Model

The threat model adopted in this paper is the same as followed by previous works [10], [11], [12], [21], [22]. The trusted computing base (TCB) contains only the processor package in this model and an adversary can access the data stored in main memory (DRAM). In such a setting, the data is communicated via address and data buses to DRAM. The data is encrypted to ensure data secrecy and integrity. Following the assumption made in [10], the adversary also has physical access to systems running client computations. Thus, an adversary can probe the off-chip memory bus to observe and modify communication between the secure processor and the memory.

The ORAM [8] cryptographic primitive makes the data access pattern to the main memory oblivious to an adversary. For instance, if a user accesses a sequence of program addresses $A = (a_1, a_2, ..., a_n)$, it will be translated to a sequence of ORAM accesses $O = (o_1, o_2, ..., o_m)$. With ORAM, an adversary while still being able to observe all the memory addresses (O) transmitted on the bus, has negligible probability to extract the logical access pattern (A). The value of o_i is exposed to the adversary and the value of a_i should be protected. By way of explanation, the ORAM physical access pattern (O) is statistically independent of the logical access pattern (A). As the adversary is assumed to be powerful, the aforementioned threat model is also valid for smart memory architectures like Hybrid Memory Cube, and High-Bandwidth Memory [6], [7].

In this paper, we focus on Path ORAM [9], which is currently the most practical ORAM scheme for limited client (processor) storage. Recent work by Wang et al. [21] focuses on making bandwidth sharing more effective when secure and non-secure applications co-run in a server setting. However, the proposed approach only allows execution of secure applications on a secure processor, and yet ensure near-peak bandwidth utilization.

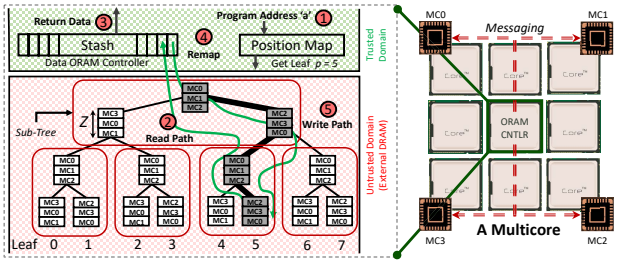

Fig. 2: *PathORAM on a multicore/multi-channel setting where ORAM controller communicates with all memory controllers. An example operation of a path access $p = 5$ is shown with sub-tree locality.*

B. PathORAM in a Multicore Setup

Path ORAM [9] functionality is based on two main hardware components, namely a *binary tree storage* and an *ORAM controller*. Binary tree stores the data content of the ORAM and is implemented on DRAM. Each node (also called a bucket) in the tree can hold up to Z useful data blocks. Usually, the ORAM capacity is set to be more than $(2\times)$ that of the DRAM size so that ORAM tree can fit into the DRAM. Remaining empty slots are filled with dummy blocks. All blocks, real or dummy, are probabilistically encrypted and cannot be distinguished. ORAM controller is a piece of trusted hardware that controls the tree structure. Besides necessary logic circuits, the ORAM controller contains two main structures, a *position map* and a *stash*. The *position map* is a lookup table that associates the program address a of a data block with a path in the ORAM tree (path p). The *stash* is a buffer that stores up to a small number of data blocks at a time. Path ORAM follows the following steps when a request on block a is issued by the processor:

1) The path (leaf) number p of the logical address a is looked up in the position map.
2) All the blocks on path p are read and decrypted, and all real blocks added to the stash.
3) Block a is returned to the processor.
4) The position map of a is updated to a new random leaf p' – ensures *unlinkability*.
5) As many blocks from stash as possible are encrypted and written on path p, where empty spots are filled with dummy blocks.

Path ORAM [11], [10] is implemented in a secure processor with multiple memory controllers. It relies on a sub-tree locality approach to achieve high memory throughput. The proposed system envisions a multicore with a single ORAM controller that communicates with numerous DRAM controllers to fetch all blocks on an ORAM path, p (c.f. Figure 2)[1]. Note that each ORAM path converts a single off-chip cache line access into $\sim$100 cache lines, thereby stressing

[1]Note that it is practically feasible to physically distribute the ORAM controller among the last-level cache banks since each distributed ORAM controller in this case will manage unique physical addresses in the main memory space. However, for simplicity of explanation we assume a single ORAM controller in this paper. The baseline Path ORAM scheme [11], [10] also implements a single ORAM controller per chip.

the available memory controllers. It packs each sub-tree with l levels together, and treats them as the nodes of a new tree. This division results in a 2^l-ary ORAM tree with $L' = \lceil \frac{L+1}{l} \rceil$ levels, where L refers to the total number of levels in the original ORAM data structure. The addressing is done by *statically interleaving* the data blocks in each sub-tree among the available memory channels. The baseline Path ORAM [11], [10] chooses a bucket size (Z) of 3 since it tends to provide best performance, whereas higher Z values incurs performance overhead due to increased dummy blocks. Contrary to this, providing lower Z values results in performance deterioration with the increased valid data bock/cache line utilization.

Figure 2 shows an example allocation of four memory controllers to the Path ORAM sub-trees. In the example, a 4-leveled ORAM tree (i.e., $L = 4$) is mapped to 4 memory controllers (i.e. $N = 4$). The ORAM tree is transformed into a shallower tree consisting of sub-trees with $l = 2$. Such transformation results in a new ORAM tree of $L' = 2$ levels. For each sub-tree, the static scheme interleaves *MC0, MC1, MC2* to the parent bucket with three cache lines (i.e. $Z = 3$), while the left child bucket is assigned *MC3, MC0, MC1*, and the right child bucket is assigned *MC2, MC3, MC0*. This approach provides a near-optimal memory space division among the available memory controllers. However, it does not ensure the path access to be equally distributed. For instance, in the above example, access of ORAM path $p = 5$ would require *MC0* and *MC2* to forward 4 data blocks to the ORAM controller, while *MC1* and *MC3* would bring 2 data blocks. Due to such load imbalance among the memory controllers the baseline scheme shows degradation in the bandwidth utilization with the increase in memory controllers – causing *starvation*. This paper proposes a PathORAM scheme that ensures proper load balancing of each ORAM path, thereby utilizing the processor's available memory bandwidth.

III. Load Balanced Path ORAM

This paper proposes a memory mapping strategy to achieve load balancing among the on-chip memory controllers. The proposed scheme builds on state-of-the-art PathORAM, and focuses on equally distributing the pool of pending ORAM requests among all memory controllers. The key idea is to distribute each individual ORAM path access (path containing regular and dummy cache lines) equally among all memory controllers.

A. PathORAM Tree Initialization

In this section, first the proposed strategy for mapping memory controllers on an ORAM tree is explained with an example, followed by a formal algorithmic description.

1) An Illustrative Example: In the example shown in Figure 3, a similar ORAM structure described in Section II is used where the ORAM tree is transformed into a sub-tree based shallower tree. However, for the proposed scheme, appropriate values for parameters Z and l need to be selected based on the number of memory channels (N) available in the system. Choosing Z and l based on these conditions allows equal distribution of a *sub-path*. Therefore, a complete ORAM path access would always be equally distributed among the available memory controllers. These conditions are as follows:

- Z is selected such that N is a multiple of Z and $Z \leq N$. For the given example, we get two values ($\{2, 4\}$) for Z. We consider $Z = 2$ for the ease of explanation.

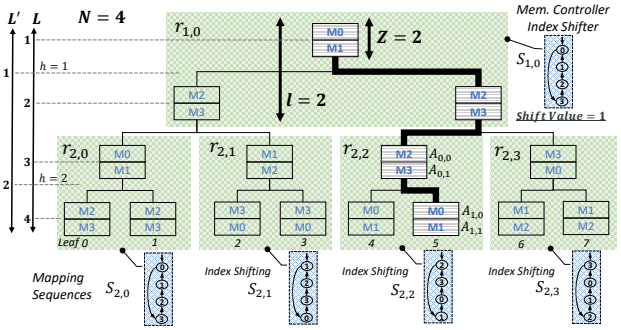

Fig. 3: *An illustrative example in which all data blocks are equally mapped on all 4 memory controllers at the parameter setting of $l = 2$ and $Z = 2$.*

- l is determined on the basis of $l * Z \leq N$ inequality, where $l > 1$. Ultimately, we get $l = 2$.

After obtaining all the parameters, the ORAM tree is divided among 2-level ($l = 2$) sub-trees, where each sub-tree contains 6 data blocks (3 buckets in total, $Z = 2$ data blocks per bucket). The resulting tree has two levels ($L' = 2$), and each sub-tree is identified by an identifier $r_{h,k}$. For $r_{h,k}$, h (a subset of L') represents the level at which the sub-tree node resides in the tree, and k denotes the position of the node from left to right at level h. To map the data blocks on memory controllers, the scheme starts by assigning data blocks of the root sub-tree node $r_{1,0}$ to available memory controllers via simple *interleaving*. Essentially, a simple ordered sequence $S_{1,0} = \{M_0, M_1, M_2, M_3\}$ of memory controllers M_i is constructed, and each block along each path of the node $r_{0,0}$ is mapped (from root to leaf) to the memory controllers. These controllers are determined by the sequence $S_{1,0}$ i.e., the first block is mapped to M_0, second to M_1 and so on, as shown in Figure 3. Similar sub-tree mapping scheme is used in case of PathORAM. However in the proposed scheme, for each sub-tree $r_{h,k}$, both child buckets of the parent get assigned the same memory controllers set. For example, in case of root sub-tree $r_{1,0}$, blocks of both right and left child buckets would be mapped on memory controller M_2 and M_3, and thereby result in equal distribution of the sub-paths among memory controllers.

Similarly, at the next level of the tree, we start from the left with node $r_{2,0}$ and a similar initial sequence $S_{2,0}$ is used to map its data blocks equally to all memory controllers – where $S_{2,0} = \{M_0, M_1, M_2, M_3\}$. In order to achieve equal memory space division among the available memory controllers, the mapping of the next node $r_{2,1}$ requires a "shifted" version of the sequence $S_{2,0}$. For this purpose, first a new parameter $ShiftVal$ is computed where $ShiftVal = \lfloor log_2(N - Z) \rfloor$, and for a particular level h, each next sequence $S_{h,i+1}$ is a logical left rotation of the previous sequence $S_{h,i}$. Note that, the $ShiftVal$ parameter uses $floor$ function to ensure that any non-integer value has been rounded down to its nearest integer. In our example, for $N = 4$ and $Z = 2$, $ShiftVal$ evaluates to 1. Therefore, $S_{2,1}$ would be the rotated version of $S_{2,0}$ i.e. $S_{2,1} = \mathsf{RoL}(S_{2,0}, ShiftVal) = \{M_1, M_2, M_3, M_0\}$. Now, the node $r_{2,1}$ is mapped according to $S_{2,1}$. Similarly, $S_{2,2} = \mathsf{RoL}(S_{2,1}, ShiftVal)$ is used to map $r_{2,2}$ and so on until the tree is mapped completely, as shown in Figure 3. The example discussed provides the following insights;

1) For any path access (path access of $p = 5$ is highlighted), the data blocks are equally distributed among the memory controllers. This is achieved via choosing Z and l parameters based on the above mentioned conditions.
2) The proposed strategy provides a near-optimal memory space division among memory controllers via the rotation scheme.

Algorithm 1 Mapping of ORAM Tree to Memory Controllers

Given:
N: Total number of memory controllers.
Z: Number of data blocks in each node of ORAM tree.
L': Levels of the new subtree based ORAM tree.
l: Number of levels in a subtree.
$ShiftVal$: Memory controller sequence rotating factor.
Algorithm:

1: **procedure** MAP_TREE (N, Z, L', l, $ShiftVal$)
2: **for** each level $h \in \{1, ..., (\mathrm{L}')\}$ **do**
3: $S_{h,0} = \{M_0, M_1, ..., M_{N-1}\}$ ▷ Initial sequence.
4: **for** each node index $k \in \{0, 1, ..., (2^l)^h - 1)\}$ **do**
5: Pick node $r_{h,k}$ and sequence $S_{h,k}$
6: MAP_NODE ($r_{h,k}, S_{h,k}$) ▷ Blocks mapping.
7: $S_{h,k+1} = \textbf{RoL}(S_{h,k}, ShiftVal)$ ▷ Rotate left.

2) Algorithmic Description: Algorithm 1 shows the structure of the proposed mapping strategy for load balancing the ORAM tree. Similar to PathORAM, the proposed scheme traverses across all the sub-trees in the ORAM tree and maps data blocks to the controllers. However, instead of statically interleaving the data blocks, the proposed scheme carefully computes the mapping sequence for each sub-tree node to ensure equal load distribution among controllers.

Given the parameters N, Z, L', l, and $ShiftVal$, each node of the transformed ORAM tree is traversed in a *breadth first* manner, starting from the root level, using the MAP_TREE() procedure. For each node $r_{h,k}$, its corresponding memory controllers sequence $S_{h,k}$ is computed through logical-left-rotations of the initial sequence $S_{h,0} = S_{1,0}$ (where $S_{1,0}$ is the root-sub-tree sequence) by $k \cdot ShiftVal$ positions. Thereafter, MAP_NODE() procedure maps the blocks of $r_{h,k}$ to the memory controllers in the order specified by $S_{h,k}$. It maps each data block i to the controller $S_{h,k}[i \mod |S|]$ in such a way that both child buckets get the same memory controller set. It is shown in Figure 3 that for each sub-tree, the child buckets have the same set of memory controllers. Notice that the "mod" operation takes care of certain parameter settings where the path length of $r_{h,k}$ is larger than $|S|$. This procedure is repeated until the whole ORAM tree is mapped to the physical DRAM space.

B. Run-Time Path Retrieval

Algorithm 2 presents a pseudo code used by the ORAM controller to generate all cache line addresses for a given path p in order to issue DRAM fetch requests to the DRAM controllers. In the example discussed in Section III-A, suppose a request to access path $p = 5$ is generated. The input to Algorithm 2 would be the bit representation of p i.e. $0b101$. Starting from the root node of the sub-tree based ORAM tree and going down to the leaf node (Line 1, Algorithm 2), for each level L_i, first the node index N_{L_i} is computed by simply taking l most significant bits of the given path p (Line 2). In the example of accessing path $p = 5$, N_{L_i} would be $0b10$. The first bit in $N_{L_i} = 0b10$ i.e., "1" implies that from the root sub-tree, the sub-path containing the *right* child bucket would be taken. Essentially, $N_{L_i} = 0b10$ or 2 shows that the sub-path for $p = 5$ lies in the 2^{nd} child sub-tree of the root.

Algorithm 2 Addressing Algorithm for the Proposed Scheme.

Given:
$ShiftVal$: Memory Controller Sequence Rotating factor.
Z: Number of Data Blocks in each ORAM bucket.
L': Levels of the New Sub-tree based ORAM tree.
l: Number of Sub-tree levels.
$\overrightarrow{M} = (0, 1, \cdots, N-1)$
Input:
p: Path to Access.
Output:
$R[|\overrightarrow{M_i}|][\,]$: Set of Requests for each Controller.
Algorithm:

1: **for** each level $L_i \in \{1, ..., (\mathrm{L}')\}$ **do**
2: $N_{L_i} = p[MSB$ **down to** $(MSB - (l-1))]$
3: s $= ShiftVal \cdot N_{L_i}$ ▷ Shift Index
4: $\overrightarrow{V} = \overrightarrow{M}$ **RoL** s ▷ Rotate $\overrightarrow{M}$ left by s positions
5: **for** each sub-tree level $j \in \{0, 1, ..., (l-1)\}$ **do**
6: **for** each block $b \in \{0, 1, ..., (Z-1)\}$ **do**
7: $R[\overrightarrow{V}(jZ + k)]$.**append**$(A_{j,b})$
8: $p \leftarrow p[(MSB - l)$**down to** $0]$ ▷ p for the next level
9: **return** R

Using N_{L_i} and the given $ShiftVal$ parameter (computed as 1 in Section III-A), a shift index s is computed (Line 3). An index vector $\overrightarrow{M}$ is "rotated" by s positions to yield another index vector $\overrightarrow{V}$ (Line 4), where vector $\overrightarrow{V}$ represents the DRAM controller mapping order for the cache lines on the sub-path common between the path p and the corresponding sub-tree node at level L_i. Hence, to access path $p = 5$, vector $\overrightarrow{M}$ is rotated twice (s $= ShiftVal \times N_{L_i} = 2$) to yield $\overrightarrow{V} = (2, 3, 0, 1)$. Finally, going through each node of the sub-tree from root to leaf on the relevant sub-path, and using the information of vector $\overrightarrow{V}$, the corresponding DRAM controller designated to each cache line $A_{j,b}$ (on Line 7) is identified and a fetch request to this cache line is appended to the appropriate vector entry of R. In case of accessing path $p = 5$, the cache line addresses $A_{0,0}$, $A_{0,1}$, $A_{1,0}$, and $A_{1,1}$ would be generated for memory controllers 2, 3, 0, and 1 respectively (shown in Figure 3). After this iteration and before moving to the next level of the tree, the l most significant bits of p are discarded (Line 10), and then the next iteration starts. Therefore, in the next iteration, $N_{L_i} = 0b1$ is computed. A single bit in N_{L_i} represents the last sub-tree level and identifies which child to be taken to access the leaf node. In the example, the bit value of 1 describes that within the 2^{nd} sub-tree, the right child bucket of the parent would be taken to access path $p = 5$. This process is repeated for each of the L' levels of the meta tree, and finally the populated requests vector R is returned. For simplicity of explanation, the details of the address offset

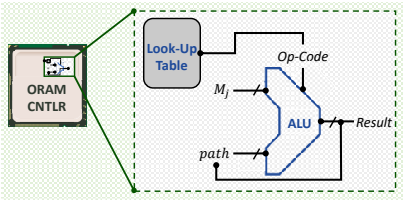

Fig. 4: *Additional hardware for the proposed ORAM scheme. Only a look-up table of N-bits is added to keep track of the memory controller sequences. The same PathORAM controller (used for* mod *operations) is used for multiplication and rotate functionalities.*

computation for $A_{j,b}$ are omitted. The root sub-tree would always be assigned with the initial sequence, $\overrightarrow{M} = (0, 1, 2, 3)$. To obtain this sequence, two extra 0 bits are padded at the start (MSB) of the N_{L_i} bit representation to yield $N_{L_i} = 0b00101$. Iterating over these padded bits results in no rotation of the sequence, and hence it helps in assigning cache lines to the root sub-tree.

Note that the proposed scheme adds a few extra operations on top of PathORAM scheme. These operations are for (1) calculating the shift index (Line 4) that involves a multiplication operation, and (2) the logical left rotate function to calculate memory controller sequences. The multiplication and rotate left operations can be done via already present ALU in the PathORAM controller for mod operations. However, to keep track of the controller sequences and to generate respective $op - codes$, a small look-up table of size $N - bits$ is required for each level of the sub-tree (c.f. Figure 4). The vector R can reside easily in the registers. The additional overheads of these operations are incorporated in the reported completion time for the proposed scheme.

IV. Discussion

This section discusses the security aspects of PathORAM and the proposed scheme. Moreover, the importance of keeping the trusted computing base (TCB) as small as possible is reviewed. Finally, the section concludes by providing insights to the applicability of the proposed scheme in a NUMA system configuration.

A. Security of the Proposed ORAM Scheme

PathORAM relies on obfuscating memory access patterns to main memory and limits an adversary from inferring secret information. Upon each off-chip (or ORAM) access, the data block of interest is retrieved from the path, and the path is stored back in the ORAM tree. In the case of only one memory controller, a single randomly accessed path is revealed to an adversary for every ORAM access. However, an adversary would not be able to gain any useful information as the path would be shuffled for the next ORAM access (c.f. Section II) – ensuring *unlinkability*. Prior work [10] has shown that PathORAM provides the same level of security guarantees even if deployed in a multiple memory controller setting, and again reveals only a single randomly accessed path. In comparison to PathORAM, the proposed scheme provides an efficient mapping strategy for load and capacity balance among parallel memory controllers to ensure near-peak bandwidth utilization. As the structure of the proposed scheme resembles that of PathORAM, it guarantees similar level of security as PathORAM.

1) Minimal Trusted Computing Base: The careful design and implementation of a system's trusted computing base is paramount to its overall security. The key challenge is to provide extensive security guarantees while keeping the TCB minimal. PathORAM successfully addresses this challenge by providing strong defenses against memory bus side channel attacks, keeping the trusted computing base minimal to the processor package only. Prior works [6], [7] provide security guarantees by using cryptographic primitives to send encrypted commands, and addresses over the untrusted memory bus. Through encryption, access patterns are cryptographically obfuscated from malicious entities to reveal confidential information. Deploying such methods eliminate the need for obfuscating memory access patterns via PathORAM due to lower performance overheads (reported as $\sim$11% overhead over an untrusted DRAM for [6], [7]). However, these works rely on expanding the TCB to main memory, which demands higher cost, and incurs greater security risks. Whereas, as mentioned before, PathORAM limits the TCB to only the processor package.

With PathORAM being memory bandwidth hungry, the proposed scheme provides a smart mapping scheme to reduce its overheads by leveraging the trends of advancements in memory technologies. In a nutshell, deploying the proposed scheme alongside PathORAM results in an efficient, and highly secure memory access pattern obfuscation scheme that provides strong security guarantees with a small trusted computing base.

2) Spatial & Temporal Path Accesses: Each algorithm has a unique memory access pattern and data request sequence which further generates different ORAM path access patterns. If the access pattern of the algorithm is such that paths are read in a spatial manner, PathORAM would not reveal any secret information as only a single path access would be revealed. Contrary to this, if the access pattern of the algorithm enforces the paths to be read in a temporal fashion then upon each access, the path would be shuffled and remapped. The property of shuffling, and remapping data blocks to new random leaf nodes by PathORAM – to ensure *unlinkability* – provides security regardless of algorithm access patterns.

3) Memory Footprint: The structure of PathORAM is such that in order to access a data block of interest, the entire path is read and the requested data block of concern is forwarded to the requesting core by the ORAM controller. Clearly, reading an entire path introduces an extensive memory footprint which can lead to potential security vulnerabilities. However, PathORAM ensures that the access patterns of any two data request sequences are computationally *indistinguishable*. Therefore, having a larger memory footprint would bring no harm to the system. With the proposed scheme following the exact same structure as that of PathORAM, it provides similar security guarantees in terms of indistinguishability.

B. Scalability to NUMA systems

This section provides a discussion on how the proposed ORAM scheme can be deployed on Non-Uniform Memory Access (NUMA) systems. The primary focus of the proposed ORAM path mapping scheme is to ensure load and memory capacity balancing for a single compute node (machine) managing its own dedicated memory. However, to employ

the proposed scheme in a NUMA system configuration, an ORAM controller would be required per node, where each ORAM controller manages and load balances its respective (local) memory. In case of a remote access, where one node requests data from another node, a fetch request is initiated by the requester node. The ORAM controller for the server node processes the memory access – in a load balanced fashion – and returns the data to the requester via a response message. Clearly, the entire aforementioned process relies on nodes communicating with each other. An adversary can infer confidential information by just monitoring the communication channel. Therefore, a secure network protocol is required for such messages to ensure data and address protection between machines.

V. Methodology

The Graphite multicore simulator [23] is modified to model the ORAM schemes for evaluation. Graphite simulates a tiled multi-core chip. The hardware configurations are listed in Table I. Eight on-chip memory controllers are considered, each providing a stand-alone bandwidth of $80GB/Sec$. DRAMSim2 [24] is used to model the internal structures of DRAM.

Splash-2 workloads [25] and OLTP database management system (DBMS) [26] workloads, namely ycsb [27], and tpcc [28] are considered to evaluate the proposed ORAM scheme against various implementations. Three graph analytic workloads – Single Source Shortest Path (sssp), Triangle Counting (tri_cnt), and PageRank (pagerank) – from CRONO [29] benchmark suite are considered. California road network [30] is used as an input to the graph algorithms. Six machine learning workloads are also considered. For Convolution Neural Network (CNN) MNIST (mnist), Multi-Layer Perceptron (mlp) and K-Nearest Neighbors (knn), handwritten digit dataset [31] is used. Another CNN benchmark, gtsrb uses German traffic sign [32] as an input. ImageNet [33] is provided as an input to two CNN benchmarks, alexnet, and squeezenet. The performance over traditional DRAM is considered as a baseline. State-of-the-art PathORAM is used for comparing the proposed ORAM scheme over multiple on-chip memory controllers. The completion times for the proposed scheme incorporate the additional latency of 4 *cycles* due to extra hardware to compute the sub-tree sequence. The default parameters for ORAM schemes are shown in Table I. Unless otherwise stated, all experiments use these default ORAM parameters.

TABLE I: *System Configuration.*

Secure Processor Configuration	
Core model	1 GHz, in-order core
Total Cores	64
L1 I/D Cache	32 KB, 4-way
Shared L2 cache	128 KB per tile, 8-way
Cacheline (block) size	128bytes
DRAM bandwidth	640 GB/s
Total Memory Controllers	8
Conventional DRAM latency	100 cycles
Modeled DRAM Configuration [24]	
Memory Type	Double Data Rate (DDR)
Memory Capacity	16 GB
Total Channels	8
Number of Banks	4
Rank Count	4
Default ORAM Configuration	
ORAM Capacity	32 GB
Working Set Size	16 GB
Number of ORAM hierarchies	4
PLB Size	256kB
Stash Size	100 Blocks
Compressed PosMap & Integrity	Enabled

VI. Evaluation

This section first compares the performance, and energy of both ORAM schemes. Then, the load distribution capabilities are discussed and compared. Lastly, the proposed ORAM scheme is compared with PathORAM in terms of performance. Parameters such as the number of memory controllers, and total memory bandwidth are increased to observe the practicality of the proposed scheme.

A. Performance Analysis

This section discusses the performance numbers for all workloads when the proposed and PathORAM schemes are applied (c.f. Figure 5:(a)). The reported numbers are normalized to the DRAM insecure baseline. From Figure 5:(a), we observe that the proposed scheme shows performance improvement over PathORAM, for all workloads. Figure 6 reports the L2 cache misses per kilo-instructions (MPKI) to observe the off-chip accesses required by these applications. A general trend has been observed across all benchmarks, i.e., an increased MPKI value correlates with higher ORAM overheads. In all such cases, the performance benefits from the proposed scheme are notable compared to the PathORAM scheme.

Splash-2 workloads, such as cholesky, radix, and fft show significantly higher ORAM overheads. This is because of the fact that these workloads incur high off-chip misses or higher MPKI values (c.f. Figure 6). The proposed scheme effectively load balances these off-chip accesses among available memory controllers and results in reducing these overheads.

In machine learning workloads, gtrsb and alexnet show high ORAM overheads due to their large model sizes that do not fit into the on-chip caches, thereby resulting in high MPKI. In comparison, squeezenet shows an immense performance improvement with the same classification accuracy of alexnet. This is mainly due to squeezenet's *inception model* that greatly reduces the model size of the classifier. A negligible performance difference in DRAM and the proposed scheme is observed for the case of squeezenet. This observation is highlighted since a secure squeezenet implementation with near optimal performance would be highly desirable in many emerging real-time systems for image classification.

For graph processing, all three benchmarks show high MPKI. This is due to the lack of locality in the input graph that offers sparse and random node connections, as well as a graph size that does not fit the on-chip caches. The high MPKI stresses the PathORAM scheme in terms of performance, leading to prohibitive $4\times$ to $6\times$ performance overheads compared to the DRAM baseline. Although the proposed scheme reduces the performance overhead, it is still 3 – $4.5\times$ higher than the DRAM baseline.

Workloads, such as several Splash-2 benchmarks (lu_c, lu_nc, volrend), as well as database benchmarks (tpcc and ycsb) do not show significant ORAM overheads due to smaller input data sets, better locality, and lower MPKI values (off-chip misses). However, the proposed scheme still offers

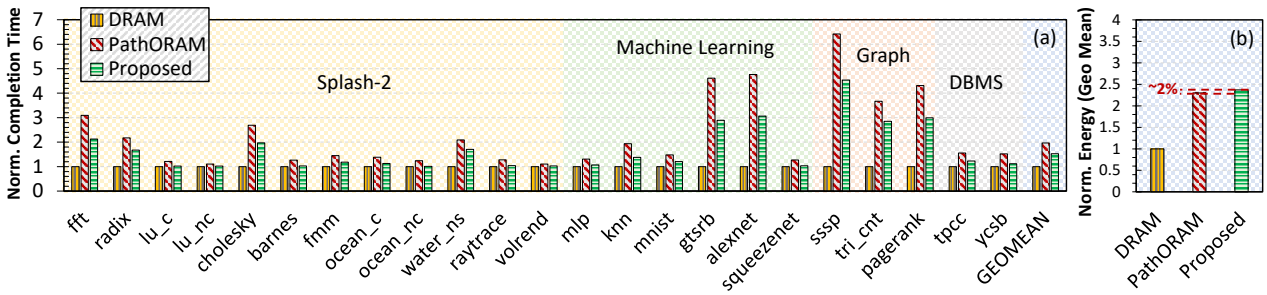

Fig. 5: *Completion times (a) are shown for all workloads normalized to the DRAM. Energy numbers for geo mean results are also presented (b). These numbers are reported for a memory bandwidth of 640GB/Sec via 8 memory controllers.*

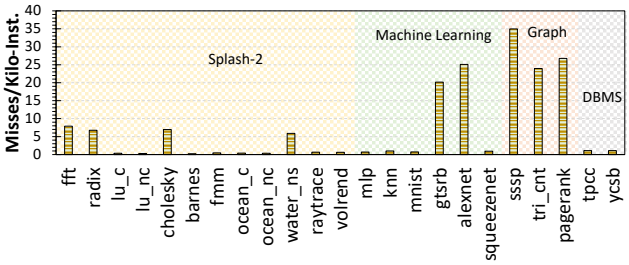

Fig. 6: *L2 Misses Per Kilo-Instructions for all workloads. Higher the MPKI, higher would be the off-chip accesses.*

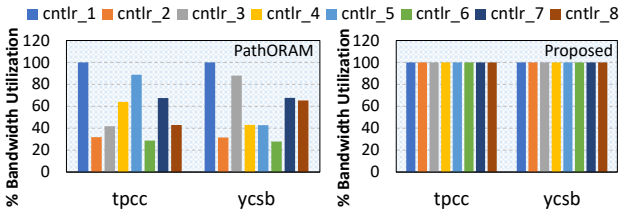

Fig. 7: *Per-controller bandwidth utilization of PathORAM and the proposed scheme. DBMS workloads are shown as an example to exhibit the load imbalancing of PathORAM.*

performance improvements for these benchmarks over the PathORAM scheme. Together, considering all workloads in Figure 5:(a), PathORAM and the proposed scheme show overheads of ∼2× and ∼1.5× respectively over the DRAM baseline. The proposed scheme provides an average performance improvement of 24% over PathORAM.

B. Energy Analysis

The dynamic energy analysis of the evaluated schemes is shown in Figure 5:(b). Geometric mean values from all benchmarks are shown, and normalized to the DRAM baseline. The proposed scheme incurs a negligible energy overhead of ∼2% over PathORAM, and an overhead of ∼2.5× over the conventional DRAM. The energy overheads of the proposed scheme over PathORAM are negligible because it reuses much of the ORAM controller to perform the proposed block mapping scheme.

C. Balanced Load Distribution

Figure 7 shows the bandwidth utilization of each memory controller with the PathORAM and the proposed scheme. Only

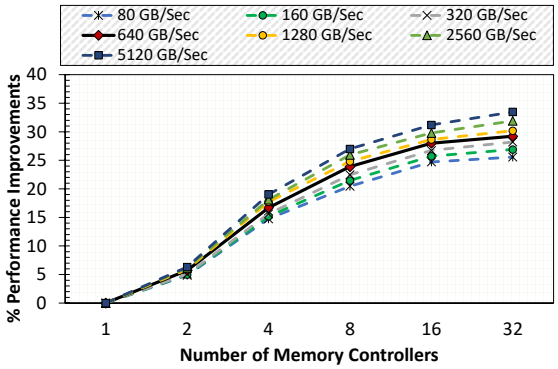

Fig. 8: *Performance improvements observed when memory controllers are increased for each bandwidth point. Bold line shows improvements for a total bandwidth of 640GB/Sec.*

DBMS benchmarks are shown for the ease of explanation. Clearly, due to load imbalance among memory controllers in PathORAM, symmetry in terms of per-controller bandwidth utilization is not observed. However, the proposed scheme ensures proper load distribution among all memory controllers, and thereby uniform peak per-controller bandwidth utilization is observed. Although not shown in the figure, other benchmarks observe similar trends, where the proposed scheme distributes load evenly among all memory controllers to achieve 100% bandwidth utilization.

D. Scalability with Bandwidth and Memory Controllers

The experiments discussed so far have targeted a system with 8 memory controllers, providing a total memory bandwidth of 640GB/Sec – each memory controller contributes 80GB/Sec. Figure 8 shows performance improvements observed when memory controllers are increased for numerous memory bandwidth points, and provides the following two observations. (1) When the on-chip memory controllers are increased, PathORAM falls short of distributing the load in an equal proportion among all memory controllers. On the other hand, the proposed scheme keeps the load distribution balanced. Therefore, performance improvements are observed to increase from ∼5.7% to ∼29.2% when the memory controllers are increased from 2 to 32. (2) For a fixed number of memory controllers, the performance improvements tend to increase at a much slower rate with the increase in total memory bandwidth. Consider the case of 8 memory controllers, the performance improvement is observed to increase from

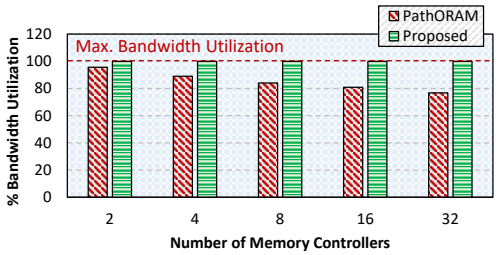

Fig. 9: *Memory bandwidth utilization with the increase in number of memory controllers.*

$\sim$24% to $\sim$27.5% when the memory bandwidth is increased from $640GB/Sec$ to $5120GB/Sec$. This performance gap is observed to be negligible considering the dramatic increase in the total memory bandwidth.

The performance improvements are observed to be less for the case of 2 memory controllers for any bandwidth point. This is mainly due to the fact that with lesser memory controllers, PathORAM is capable of distributing load in nearly the same fashion as done by the proposed scheme (Prior works [10], [11] have shown a bandwidth utilization of $\sim$96% with 2 memory controllers). However, the proposed scheme scales Path ORAM as future many-core processors integrate a larger number of on-chip memory controllers.

E. Memory Bandwidth Utilization

Figure 9 shows the average memory bandwidth utilization of both proposed and PathORAM schemes with the increase in memory controllers. It is evident from the figure that as the memory controllers increase, PathORAM scheme falls short of utilizing the total memory bandwidth due to inefficient load distribution. The bandwidth utilization of the PathORAM scheme degrades from $\sim$96% to $\sim$76% when the memory controllers are increased from 2 to 32. However, the proposed scheme utilizes the entire memory bandwidth by distributing load evenly among all memory controllers.

VII. Conclusion

This paper discusses the impact of load balancing and bandwidth utilization for ORAM performance overheads in a multiple memory controller setting. A novel ORAM path distribution scheme is proposed that ensures load and memory capacity balance among available multiple on-chip memory controllers. It enables each memory controller to operate at its peak bandwidth that ensures reduction in ORAM overheads. Results show an average performance improvement of 24% over state-of-the-art PathORAM scheme when a secure multi-core processor implements 8 on-chip memory controllers with a total memory bandwidth of $640GB/Sec$. These performance numbers improve when the system enables a larger number of memory controllers. By doing so, the proposed scheme consistently outperforms the PathORAM scheme in terms of load distribution.

Acknowledgment

This research was partially supported by the National Science Foundation under Grants No. CCF-1550470 and CNS-1413996.

References

[1] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, " AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *ACM ICS*, 2003.
[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ ISCA*, 2013, p. 10.
[3] M. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *Network and Distributed System Security Symposium (NDSS12)*, 2012.
[4] X. Zhuang, T. Zhang, and S. Pande, "HIDE: an infrastructure for efficiently protecting information leakage on the address bus," in *11th ASPLOS*, 2004.
[5] T. M. John, S. K. Haider, H. Omar, and M. V. Dijk, "Connecting the dots: Privacy leakage via write-access patterns to the main memory," *IEEE Transactions on Dependable and Secure Computing*, 2017.
[6] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," *SIGARCH Comput. Archit. News'17*.
[7] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th ISCA'17*. ACM.
[8] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," in *J. ACM*, 1996.
[9] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *ACM CCS*, 2013.
[10] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S.Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *ISCA'13*.
[11] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Design and implementation of the ascend secure processor," *IEEE Transactions on Dependable and Secure Computing*, vol. PP, no. 99, pp. 1–1, 2017.
[12] C. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, "Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram," in 20^{th} *(ASPLOS)*, 2015.
[13] Intel, "http://ark.intel.com/products/75799/intel-xeon-phi-coprocessor-7120p-16gb-1_238-ghz-61-core," 2014.
[14] "https://www.nvidia.com/en-us/geforce/products/10series/titan-xp/."
[15] Wikipedia, "Xeon phi 7120p." [Online]. Available: https://en.wikipedia.org/wiki/Xeon_Phi
[16] M. O'Connor, "High-Bandwidth Memory (HBM) Highlights." [Online]. Available: https://www.cs.utah.edu/thememoryforum/mike.pdf
[17] R. Hadidi, B. Asgari, J. Young, B. A. Mudassar, K. Garg, T. Krishna, and H. Kim, "Performance implications of nocs on 3d-stacked memories: Insights from the hybrid memory cube," *CoRR'17*, vol. abs/1707.05399.
[18] J. Schmidt and U. Bruning, "openhmc - a configurable open-source hybrid memory cube controller," in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015.
[19] P. Rosenfeld, "Performance Exploration of the Hybrid Memory Cube," PhD thesis, University of Maryland, 2014. [Online]. Available: https://drum.lib.umd.edu/handle/1903/15372
[20] NVidia, "Increasing trends in memory bandwidth and throughput," 2017. [Online]. Available: https://devtalk.nvidia.com/default/topic/985255/theoretical-real-shared-dram-peak-memory-throughput/?offset=10
[21] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-oram for effective memory bandwidth sharing in server settings," in *2017 IEEE HPCA*.
[22] S. K. Haider, H. Omar, I. Lebedev, S. Devadas, and M. van Dijk, "Leveraging hardware isolation for process level access control & authentication," in *SACMAT' 17*.
[23] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *HPCA*, 2010.
[24] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE CAL*, Jan 2011.
[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd ISCA*, 1995, pp. 24–36.
[26] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, 2014.
[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC'10*.
[28] The Transaction Processing Council, "TPC-C Benchmark (Revision 5.9.0)," http://www.tpc.org/tpcc/spec/tpcc_current.pdf, June 2007.
[29] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*.
[30] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.
[31] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.
[32] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The german traffic sign recognition benchmark: A multi-class classification competition," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, July 2011, pp. 1453–1460.
[33] J. Deng, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, June 2009, pp. 248–255.