LAORAM: A Look Ahead ORAM Architecture for Training Large Embedding Tables

Rachit Rajat*
Yongqin Wang*
rrajat@usc.edu
yongqin@usc.edu
University of Southern California
Los Angeles, California, USA

Murali Annavaram
annavara@usc.edu
University of Southern California
Los Angeles, California, USA

ABSTRACT

Memory access patterns have been demonstrated to leak critical information such as security keys and a program's spatial and temporal information. This information leak poses a significant privacy challenge in machine learning models with embedding tables. Embedding tables are used to learn categorical features from training data. The address of an embedding table entry carries privacy sensitive information since the address of an entry discloses features associated with a user. Oblivious RAM (ORAM), and its enhanced variants, such as PathORAM, have emerged as viable solutions to hide leakage from memory access streams. PathORAM fetches an entire path of memory blocks for every memory fetch request, thereby leading to substantial bandwidth and performance overheads.

In this work, we present Look Ahead ORAM (LAORAM), an ORAM framework designed to protect user privacy during embedding table training. LAORAM exploits the unique property of ML training, namely the training samples that are going to be used in the future are known beforehand. LAORAM preprocesses the training samples to identify the memory blocks which are accessed together in the near future. LAORAM combines multiple blocks accessed together as superblocks and tries to assign all blocks in a superblock to few paths. Thus, future accesses to a collection of blocks can be satisfied from a few paths, effectively reducing the number of reads and writes to the ORAM. To further increase performance, LAORAM uses a fat-tree structure for PathORAM, i.e. a tree with variable bucket size, effectively reducing the number of background evictions required, which improves the stash usage. We have evaluated LAORAM using both a recommendation model (DLRM) and an NLP model (XLM-R) embedding table configurations. LAORAM performs 5 times faster than PathORAM on a recommendation dataset (Kaggle) and 5.4 times faster on an NLP dataset (XNLI) while guaranteeing the same security guarantees as the original PathORAM.

*Both authors contributed equally to this research.

ISCA '23, June 17–21, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0095-8/23/06.
https://doi.org/10.1145/3579371.3589111

CCS CONCEPTS

• **Security and privacy → Security in hardware**; • **Computer systems organization → Architectures**; • **Computing methodologies → Machine learning**.

KEYWORDS

Memory, ORAM, Security, Recommendation Systems, Embedding Tables

ACM Reference Format:
Rachit Rajat, Yongqin Wang, and Murali Annavaram. 2023. LAORAM: A Look Ahead ORAM Architecture for Training Large Embedding Tables. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579371.3589111

1 INTRODUCTION

Privacy of user data, particularly in cloud resources, is becoming critical. In the context of machine learning, data owners want to keep their data confidential when using cloud GPU instances like Amazon EC2 P4d, and Microsoft ND A100 v4 series virtual machines to train their models. Privacy-preserving machine learning in the cloud has thus gained prominence in recent years. Researchers have explored different approaches to privacy protection in machine learning. Many prior works protect the data and/or model parameters from being visible to an adversary. These approaches include Homomorphic Encryption [16, 18, 25, 29], Secure Multi-Party Computing (MPC) [31, 32, 41, 45, 48, 49], Differential Privacy [5, 17] and using Trusted Execution Environments [21, 26, 34, 43]. With the emergence of confidential computing on GPUs [1, 23, 44], which forbids unauthorized accesses to GPU memory, including those from the host OS, users could train a model with revealing their data content. Protecting data and computations is, however, insufficient for many machine learning workloads that train embedding tables.

What do Embedding Tables Reveal? Embedding tables are increasingly used at the core of multiple high-valued machine learning models, such as recommendation models [24, 35, 46] and Transformer-based natural language processing (NLP) models [13, 15, 27, 30]. Embedding tables are used to learn a representation for categorical data. Embedding tables convert semantically similar categorical items into similar embedding vectors in a multi-dimensional feature space represented by the embedding table width.

Consider an example training sequence shown on the left of Figure 1. The figure shows an embedding table that captures several different types of videos that users watch. In this example, the first training sample shows that a user watches *comedy* movies the most. The second sample shows a user who mostly watches *political* movies. The next user watches *thrillers* and so on. Let's assume users who watch comedies the most will likely watch superhero movies. After embedding table training, embedding table entries representing those two categories are likely to be similar.

In ML models that use embedding tables, knowing the embedding table entry being accessed, which is a memory address, can compromise privacy. In the above example, each of the movie categories corresponds to a specific embedding table entry. Hence, knowing the embedding table entry can compromise the user's movie preference. While a user's movie preferences may look innocuous, this example can easily extend to more challenging situations where a user's political affiliation or web surfing behavior can be extracted from knowing the embedding table entries. For instance, in recommendation systems, embedding table entries capture a user's prior behavior, such as advertisement-clicking choices and browsed news feeds. In the case of NLP models, each embedding entry may be associated with a learned representation of a word. The addresses associated with accessing categorical data in an embedding table reveal much about the user. Thus, the new generation of ML models requires protection against memory address leakage. A detailed study of attacks on embedding tables can be found in [22].

Embedding Table Training in the Cloud: Embedding tables are extremely large and are generally accessed very sparsely during any training batch. Hence, many training systems store large embedding tables in untrusted CPU DRAM to reduce GPU memory usage [9, 54]. In this training mode, GPUs fetch only a small subset of embedding table rows as needed for a training batch. For instance, in cDLRM [9], CPU memory is used to store the embedding tables of a recommendation model. GPUs are used to perform computation-intensive MLP operations. For each training batch, the GPU only receives a few embedding entries from CPU memory and places them into GPU memory, and trains on that batch. In this process, memory access requests are issued to the embedding tables on CPU memory, which is the source of information leakage.

A training sample for embedding tables consists of t embedding table entries that must be accessed. The embedding table lookup for a training sample is defined as $R = A \times W$, where W is the embedding table and $A = [e_1, e_2, ..., e_t]$. Each e_i is a one-hot vector representing categorical inputs to embedding tables. As the size of the embedding table is enormous (terabytes) [54], practically, embedding table accesses are implemented as fetching embedding entries corresponding to each e_i. The entry number of an accessed embedding table entry e_i is a valuable training sample that holds sensitive information.

1.1 Memory Address Leakage Mitigation

One popular mitigation technique for hiding memory access patterns is Oblivious RAM (ORAM)[19]. Conceptually, ORAMs access many data blocks for every memory access request to obfuscate the actual data address. Furthermore, ORAM reshuffles data placement so that future accesses to the same address will generate a different random set of addresses on the bus. ORAM guarantees that the adversary cannot distinguish between two memory accesses. Recent enhancements to ORAM design include Path ORAM [42], where the data blocks are stored in the form of a tree, and each data block is assigned a path (i.e. a leaf node). On a data block access, not only the requested data block is read but also all the blocks along the path from the root node to the leaf node are read (more in-depth detail in the background section). Unfortunately, the security guarantees of nearly all the ORAMs [10, 19, 38, 42] come with significant overhead. Theorem 1.2.2 in [19] indicates that when accessing t data blocks of a RAM which contains N data blocks, additional $max\{N, (t-1) \cdot log(N)\} - t$ data blocks are fetched in ORAM. Even the optimized PathORAM[42] needs to access $t \cdot [log(N) - 1]$ data blocks for each block accessed. These additional accesses cause memory access delays and bandwidth contention. Besides memory access delays, ORAM metadata management such as stash management, position map lookups, and data eviction logic add additional overheads for each ORAM access.

Proposals have been made to reduce the overheads of PathORAM, such as static or dynamic superblocks, as described in PrORAM [51]. Superblocks reduce the number of paths that need to be read by coalescing multiple memory accesses to be accessed by the application into a single path. Instead of reading n paths for n data accesses, a *perfectly* formed superblock of size S needs to read n/S paths from ORAM. Prior techniques strive to create such superblocks [39] by merging adjacent data blocks into bigger blocks. For example, PrORAM [51] groups **adjacent** blocks that were previously accessed together into a few superblocks with the hope that these blocks will likely be accessed together in the future.

1.2 Exploiting Lookahead for ORAM Efficiency

Looking at the past accesses to form the superblocks is very challenging for embedding table training systems. Figure 2 plots the embedding table entries accessed at each training sample for the first 10,000 training samples for the Kaggle dataset, which is a representative dataset for the DLRM recommendation model by Meta Inc [35]. The data shows that most accesses to embedding table entries are random, and only a narrow black band at the bottom of the figure illustrates that a few indices are accessed repeatedly. In the absence of good predictability, PrORAM performs similarly to PathORAM. Our quantitative comparison results presented later in this paper also demonstrate this observation.

Lookahead Property: Unfortunately, the embedding table accesses across different training samples exhibit very little predictability, rendering history-based superblock formation mechanisms ineffective. However, machine learning training exhibits a unique property. The training samples used for the several subsequent batches are known beforehand. Although predicting the future based on the past may be difficult, peeking into the future is feasible with training data. In a machine learning training pipeline, the training batches can be generated ahead of time from the training data. It is not a unique requirement for our work but a regular practice to generate training batches ahead, which we will exploit in our work [4]. In machine learning training, data points from a dataset are randomly fed to ML models. However, this randomness is predetermined by a random seed. Consequently, the sequence of data points fed to the models is determined in advance and follows a predetermined pattern. Thus, future data access pattern is known

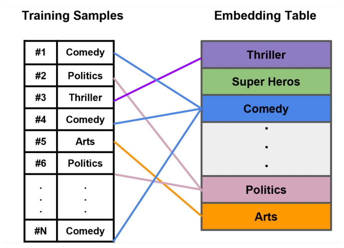


Figure 1: Left: N training samples; Right: an embedding table representation

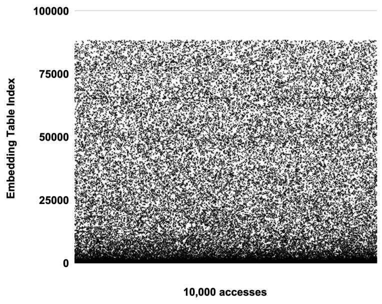


Figure 2: 10000 accesses to embedding tables for Kaggle Dataset.

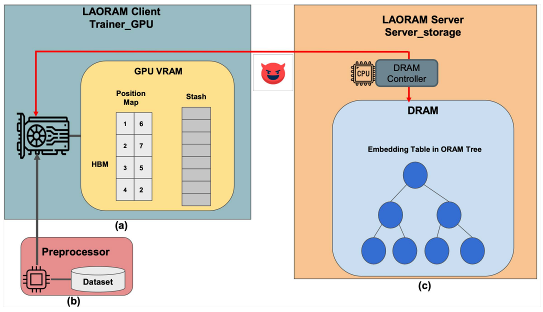


Figure 3: LAORAM Architecture Overview; black solid lines are secure communication channels and the red solid line is an insecure communication channel.

beforehand. Peeking into the future allows our Look Ahead ORAM (LAORAM) approach to form superblocks more aggressively. Unlike PrORAM, LAORAM takes advantage of this future knowledge to improve the performance of address-hiding schemes.

No alternate protection strategies: Address-based attacks cannot be mitigated by TEEs or adding differential private (DP) noise. Neither Intel SGX TEE nor ARM TrustZone offers protection to memory access patterns, and OS still manages page tables for TEE memory [14]. Adding DP noise to the memory addresses requested is impractical since that is equivalent to accessing random addresses that are unrelated to the embedding table entries that are being trained. Thus, more robust mitigation is needed as DP essentially gives computational privacy but does not protect against memory access pattern leakage.

1.3 Our Contributions

1. Dynamic superblock formation based on lookahead: In this paper, we propose LAORAM, a dynamic superblock formation technique that takes advantage of *future* access patterns. In LAORAM, the data blocks accessed together in upcoming training batches are combined to form superblocks. The future access patterns are discerned by using a low overhead preprocessing of the training samples to look ahead into the future training batches. This look-ahead dynamic superblock scheme reduces the number of reads/writes required, speeds up the memory access latency, and reduces the bandwidth requirement.

2. Fat-tree structures for reduced stash demand: Besides the usage of superblocks, we propose using fat-tree ORAM organization to mitigate the increased stash usage due to superblocks. Stash is a limited memory within the ORAM client, and accesses to the stash memory are assumed to be invisible to an adversary in all ORAM designs. Our analysis showed that if the superblock size is more than two blocks, the stash size grows rapidly. ORAM must have sufficient empty blocks along the accessed path for the superblock to be placed back into the tree. Whenever the empty space is not available on a path the block gets temporarily held back in the stash. And this probability of holding back the data block in the stash increases with superblock size. We propose a unique fat-tree organization with variable bucket sizes in the ORAM tree to mitigate growing stash size.

We have evaluated LAORAM using both a recommendation model (DLRM) and an NLP model (XLM-R) embedding table configuration. LAORAM performs 5 times faster on the recommendation dataset (Kaggle) and 5.4 times faster on an NLP dataset (XNLI).

1.4 Scope of the LAORAM

Our work applies to machine learning training systems that use embedding tables, such as recommendation systems and NLP models. These models are the enablers of growth in many industries. For instance, nearly 80% of the inference cycles in current data centers are spent in recommendation models [12]. Embedding table based NLP models power nearly all the voice-enabled smart speakers in the world. And, nearly 70% of the smart speaker users use voice assistants to access network resources. [2]. Similarly, current video streaming services are no longer video-on-demand systems; instead, they are videos automatically recommended to the users [3], which in turn is driven by embedding tables. Thus protecting the privacy of embedding table access patterns is an essential problem for the industry.

2 SYSTEM SETTING AND THREAT MODEL

2.1 Training Steps

Figure 3 provides an overview of training ML models such as recommendation systems that use large embedding tables. In these ML models, each training batch consist of a set of embedding table indices that are being learned. For instance, the training sample may include the list of indices where each index refers to an item that a user has browsed or purchased in the recent past. The goal of the training process is to learn the embeddings of these accessed items so that items browsed together or purchased together may be closer in the embedding space. Training these embedding tables is done using three components: *a) Trainer_GPU*, *b) Preprocessor*, and *c) Server_storage*. In Step 1, the trainer_GPU receives LAORAM superblock management metadata for many future training batches.

In particular, this information includes the list of embedding table indices that are going to be accessed in the upcoming training batches. In our design, this metadata is provided by a preprocessor module, which we will describe shortly. As noted earlier, when embedding tables are large, they can't fit inside the GPU's VRAM. Instead, our system stores the embedding table in a CPU DRAM. Further, to protect against address leakage, our system uses PathORAM [42]. We refer to the PathORAM-based embedding table storage as the Server_storage. In Step 2, the trainer_GPU issues a set of ORAM access requests corresponding to the embedding table entries needed by the training samples. In step 3, the Server_storage will fetch the corresponding paths from the ORAM tree and forwards them to the trainer_GPU. In step 4, the trainer_GPU then caches the ORAM paths inside its VRAM. It then performs the gradient updates to the embedding entries. The updated entries are then pushed back to the Server_storage at the end of the training step. Steps 2-4 are repeatedly executed as superblock metadata continues to arrive from the preprocessor.

2.2 System Components

The components that participate in our training steps described above can be mapped into traditional ORAM components as follows:

ORAM server: Server_storage is equivalent to the storage in PathORAM, and address accesses to this storage are considered insecure. Thus, server_storage that stores the PathORAM tree of the embedding table (and the associated host CPU) is treated as the *ORAM server* in the traditional ORAM terminology [19, 42].

ORAM client and Stash: The trainer_GPU perform computationally intensive operations of recommender systems and NLP models. The trainer_GPU issues embedding table access requests to the Server_storage. The trainer_GPU will act as an ORAM client in our system setting. It will store the ORAM management data (stash and position map) and superblock metadata received from the preprocessor inside its VRAM.

Preprocessor: The preprocessor is a special application thread that could run on the secure GPU client or any other trusted computing base. It preprocesses the training sample data and generates metadata that indicates which embedding entries will form superblocks with those entries in the incoming training samples. This component is unique to LAORAM design as preprocessing future access patterns is not practical in application domains targeted by prior ORAMs.

2.3 Threat Model:

LAORAM focuses on two adversaries. The first one is **an adversary who can snoop on the bus connected to DRAM on the Server_storage** and observe the embedding table addresses being accessed. Such an adversary could have physical access to the CPU host, where they can place a bus probe to extract address sequences. The second adversary is **a curious OS**. The curious OS can employ many state-of-art attacks to observe the data traffic between the trainer_GPU and the server_storage (we will describe one of those attacks in the next section). It means the curious OS can know the address requests issued by the trainer_GPU to the server_storage (see the red line in Figure 3). We assume the adversary cannot probe the bus between the GPU computation unit and VRAM. State-of-art GPUs use HBM as VRAM, which is tightly integrated into the GPU packages, and probing the bus between them will essentially destroy the fabric. With the above assumptions, it is LAORAM's responsibility to provide an efficient memory address obfuscation mechanism in the trainer_GPU such that memory access patterns from the trainer_GPU to the server_storage are computationally indistinguishable from random accesses.

2.4 An Example Attack on Embedding Table Addresses

In our system setting, a curious host OS on the CPU might utilize a combination of page faults ([14] Section 6.6.7) and cache side-channel [50] to obtain embedding table access patterns at a cache line granularity. For instance, the curious OS can set all the present bits of embedding table memory pages to 0. Consequently, a page fault is triggered every time the embedding table is accessed. The OS can know the page number accessed from the information provided to the page fault handler. After handling the page fault, the OS can use cache side-channel attacks to get the accessed address at a cache line granularity. Thus, a curious OS can compromise input confidentiality by using address-based attacks.

3 BACKGROUND

3.1 Embedding Tables in the Host Memory

Large embedding tables can be trained on a variety of computing platforms. For instance, one way to train the DLRM (a huge recommendation model from Meta) [35] is to distribute the embedding tables across many GPUs [33]. Since each GPU has limited memory, the model must be distributed across hundreds of GPUs. Recent work [9, 54] showed that in these distributed training systems, GPUs are vastly underutilized. Using many GPUs primarily to store the model in their memory is expensive. Hence, there is a trend toward keeping large embedding tables in the host memory, where the GPU asks the host to fetch the embedding tables from DRAM and uses fetched entries for training. Accessing the embedding tables in the host memory is a potential source of address leakage because a curious host OS can know memory access patterns to embedding tables. Thus, protecting embedding table accesses during training is an important problem.

3.2 PathORAM

ORAM [19] uses the notion of a client and server. The data is stored on the server memory, and the client requests data from the server. ORAM is the framework that helps the client hide memory access patterns from adversaries in the server. PathORAM is an implementation of the ORAM with two major components: the binary tree storage, which is on the server side, and the ORAM stash controller, which is on the client side. The stash controller consists of small trusted storage called stash and a position map data structure.

Binary Tree Storage: The data blocks in PathORAM are stored in a binary tree. Each node can hold up to Z data blocks (bucket size). Of these Z data blocks, a subset of blocks contains real user data, while the rest contains dummy data. The root node is considered level 0, and leaf is at level L. Each leaf has a path number associated with it.

Client Side: The client consists of a stash and a position map. The stash is small storage that holds data if the data blocks cannot be written back into the binary tree storage due to limited empty block availability within a given path. The position map is a lookup table that maps each real data block to one of the paths in the binary tree.

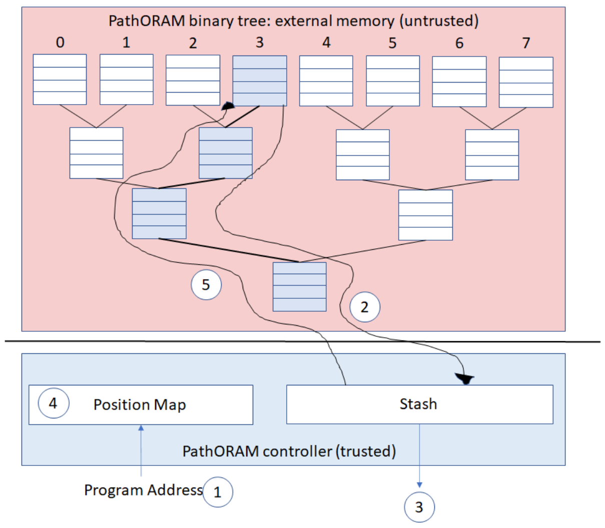

PathORAM binary tree: external memory (untrusted)
0
1
2
3
4
5
6
7
5
2
4
Position Map
Stash
PathORAM controller (trusted)
Program Address
1
3

Figure 4: PathORAM with 4 levels, and access to path 3

Figure 4 illustrates the various steps involved in accessing data in PathORAM.

(1) The client generates a request to a data address. The controller looks up the position map to identify the path where the requested data block is present in the binary storage. Let us assume that the path name is S. (2) This path name S is sent to the server, and all the buckets along the path S, from the root node to the leaf node, are sent to the client. The client will add those blocks to the stash. (3) The client then performs whatever operation is necessary on the requested data block. (4) The controller then assigns a new path S' for the accessed data block. It then updates the path associated with the data block to S' in the position map. This new path is chosen uniformly from all the leaf nodes. (5) Finally, the controller writes the blocks in the stash that can be placed along the same path which was read, i.e., path S. Any blocks in the stash that cannot be placed in path S are left in the stash.

The randomization of paths after every read/write is the foundation for PathORAM's ability to obfuscate address accesses, as the adversary sees paths are uniformly accessed on each data access.

3.3 PrORAM

While PathORAM provides obliviousness, every access requires reading an entire path and metadata management, causing a substantial bandwidth increase. PrORAM is a further enhancement over PathORAM using superblocks. A superblock is a combination of multiple data blocks with the condition that all the data blocks in the superblock are assigned the same path. PrORAM takes advantage of *past data locality* and forms superblocks from multiple data blocks. Two different approaches are presented in PrORAM: **Static Superblocks:** Static superblocks are formed by treating n consecutive data blocks as one superblock. This approach is based on the premise that spatial locality exists across nearby blocks. **Dynamic Superblocks:** When two data blocks are accessed together, a counter associated with the block pair is incremented. Once the counter goes above a threshold, the data blocks are grouped into a superblock. If the blocks within a superblock are not accessed together, the counter value is decremented. Once the value goes below the threshold, the superblock is broken back into individual data blocks. Thus superblocks are formed by coalescing blocks that are accessed together in the past, with the assumption that past history is a good predictor for future accesses. By assigning the blocks within a superblock to a single path the intuition is that more useful blocks can be fetched from each path.

3.4 Stash Management

The stash is small storage on the client side, used to buffer the data when a particular path is read from the binary tree storage. When the data block has been accessed, the path associated with the data block is changed, and the stash is written back along the same path which was read. This approach forces some data blocks to be left in the stash if there is no space left on the write path to write all the blocks in the stash that are assigned to that path. Excessive occurrence of this phenomenon can cause the stash to overflow. Background eviction has been used in PathORAM and PrORAM to reduce the stash size. Background eviction issues dummy reads to random paths once the stash occupancy goes above a certain threshold. Dummy reads do not change any blocks' associated path and help move blocks on the stash that are assigned to these paths. Thus, dummy reads reduce the number of blocks in the stash.

Another way to reduce stash overflow is to leave dummy blocks along each path, so there is a higher probability of finding an empty block in a path. For instance, PathORAM and PrORAM implementations suggest a tree structure with bucket size Z to have Z-1 dummy blocks in every node, but the space overhead of such an approach is large.

4 LAORAM DESIGN

ML model training has the unique advantage that the future memory access patterns to the embedding tables are known beforehand because the embedding table indices that will be accessed by upcoming training samples can be analyzed beforehand. LAORAM exploits this knowledge in forming superblocks. As explained earlier, superblocks are multiple data blocks grouped together such that all the data blocks in the superblocks are assigned the same path.

LAORAM merges embedding table entries that will be accessed in the upcoming training batches into superblocks. Reads and writes in LAORAM happen at the granularity of superblocks. If the superblock size is four, a single access to the superblock accesses all four of them. Once the superblock is accessed, the path of all four data blocks is changed independently based on their future locality. Hence, the fact that the group of blocks is formed into a superblock during one access has no bearing on how those blocks are paired in the future. For instance, if all four blocks are going to be accessed separately with different data blocks in the future, their future paths would be different. If only two of the four blocks are going to be accessed together in the future, then only those two blocks are placed together in a superblock. The other two would be in separate superblocks.

The training procedure used by LAORAM is already described in Section 2. We now describe the overall architecture and the detailed algorithmic design of the LAORAM.

4.1 LAORAM Architecture

4.1.1 LAORAM Server(Server_storage): The LAORAM server, referred to as server_storage, is the CPU DRAM storing embedding table entries in a binary tree ORAM structure. It is equivalent to a traditional ORAM server. The LAORAM server is responsible for providing the embedding table entries requested by the GPU trainer. As noted earlier, accesses to the CPU DRAM are considered insecure, and an adversary can observe the address patterns to the server_storage. Hence, all accesses sent to the CPU DRAM on the LAORAM server must be obfuscated using the path oblivious scheme of LAORAM.

4.1.2 LAORAM Client. **The trainer_GPU** uses its VRAM to store the position map and stash memory, along with the cached embedding entries that are trained on the GPU, and it performs gradient descent operations on the cached embedding table entries to converge to the final ML goal. The position map and stash serve their traditional roles as in PathORAM. In particular, the position map stores the path where each memory block (an embedding table index) is stored in the ORAM tree. The stash stores all the data blocks that cannot be evicted to the server_storage. The trainer_GPU will use those data structures and metadata received from the preprocessor to ask the host CPU to fetch embedding table entries it needs at a given time and store them in the VRAM. In LAORAM, instead of requesting the actual embedding table entries, the trainer_GPU will request the corresponding paths associated with those embedding table entries to prevent leakage.

The preprocessor, a special LAORAM client component not present in the traditional ORAM designs, is responsible for preprocessing upcoming training samples and extracting embedding table entries that are going to be accessed together to form superblocks. It will also assign a random path to each newly formed superblock. Information about superblock formations and the random path associated with each superblock are communicated to the trainer_GPU. The trainer_GPU will assign the paths to data blocks based on the information received from the preprocessor and updates its position map. The GPU then issues read requests to all the paths associated with the embedding entries in the upcoming training batch and caches them locally before starting the batch training process.

4.2 Preprocessing Algorithm

4.2.1 Algorithm overview. This subsection describes the preprocessing algorithm. Preprocessing consists of two steps: 1) Dataset scan, and 2) Superblock path generation.

4.2.2 Dataset scan. The preprocessing algorithm needs the superblock size S as the input. The preprocessing node looks at the upcoming training batches and places the next S entries into a superblock bin. It continues this binning process to create as many bins as it can preprocess while staying within the compute and memory limitation of the preprocessing node. For instance, the preprocessing node may scan an entire epoch of training batches if it has sufficient memory capacity to create and store all the bins necessary to hold the entries within the epoch.

4.2.3 Superblock path generation. In this step, the preprocessor will pick a random path for every superblock bin formed during the scanning step. Paths are chosen from a uniform distribution of $U(1, L)$ where L is the number of leaves in the PathORAM. That is to say, a path for each superblock bin is chosen uniformly from among one of the leaf nodes. Thus each superblock bin is now assigned a path number. After each superblock bin gets its path, the preprocessor will produce a superblock to future path number mapping. This (superblock, future path numbers) metadata is then securely transmitted to the trainer_GPU. The trainer_GPU stores this metadata to assign predetermined future paths to data blocks when accessed.

5 FAT TREE FOR STASH EFFICIENCY

Increased Stash Usage using Superblocks: One potential challenge with using superblock designs in ORAM is stash growth. With superblocks, multiple blocks are going to be placed in the same path. Hence, when the trainer_GPU needs to write the data back to the server_storage it has to find sufficient space along a given path to accommodate all the blocks within a superblock. When some of the blocks in the superblock cannot be accommodated in the given path those blocks must be temporarily held in the client's stash memory. By forcing a single path assignment to all the blocks within a superblock, the probability that a block must be held in the stash increases. When the stash is full, trainer_GPU can issue dummy evictions (read random paths without accessing any data block) to drain the stash. These dummy reads will degrade performance but are the only way traditional ORAMs can reduce stash usage.

Naive Increase Bucket Size: One solution to this problem is to increase the bucket size of the nodes in the tree, which will reduce the stash size. A larger bucket size essentially allows for more opportunities to place a block in a given path, thereby increasing the probability of stash eviction. However, such a naive solution does not efficiently utilize added memory spaces.

Key Observation about Buckets in Different Levels: One key observation we have made is that the probability of a data block being written in a particular level goes down as we go to the leaf node in the tree. In particular, the probability of a block being written back to the root is 0.5, the probability of a block being written to a level 1 node is 0.25, and so on. With this insight, we propose to use a variable bucket size with wider buckets near the root and narrower buckets near the leaf. This structure draws parallels to the fat-tree network topology [28], as shown in Figure 5. For example, when the original bucket size is 5, the bucket size at the root will be 10 and it decays to a bucket size of 5 near the leaf.

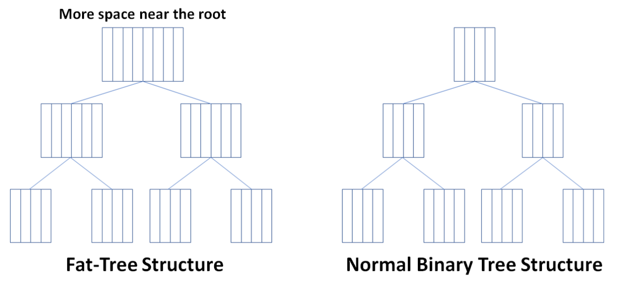


Figure 5: Fat-tree vs normal binary tree. Fat-tree has more buckets near the root and hence has lower chances for data blocks to be left in the stash.

Fat-tree organization: In our fat-tree implementation, bucket sizes are tied to the leaf bucket size. If the leaf bucket size is x then the bucket size at the root node is $2x$. For example, if the leaf node bucket size is 5 and the number of levels in the tree structure is 6, the root (level 0) will have a bucket size of 10, level 1 will have a bucket size of 9, and so on until the leaf level has a bucket size of 5. Ideally, the bucket size should grow exponentially as the level gets closer to the root to provide the same probability of finding an empty space at each level. However, exponential growth is not practical in ORAM tree organization due to huge overheads at the root. Thus, we choose linear growth. Despite the fact that our bucket growth is not exponential, our fat-tree design is able to improve the utilization of memory spaces at each tree level to accommodate data blocks sent from the stash without causing undue stash growth.

Compared with increasing bucket size for the entire tree uniformly, our fat-tree structure can trigger fewer evictions while slightly increasing the memory requirement for wider buckets near the root. Wider buckets near the root give more opportunity for data blocks to be written back to the server_storage, and hence the fat-tree structure uses memory space more effectively. As we show later, the fat-tree triggers 12.4% fewer evictions while using 16.6% less memory space compared to doubling the bucket size uniformly.

6 SECURITY ANALYSIS

Each subsection below presents the security analysis of the path obliviousness of LAORAM components.

6.1 ORAM Server (server_storage) Access Patterns

This section will discuss the obliviousness of memory access patterns to the LAORAM server (server_storage) where embedding tables are stored. Server_storage will read data from CPU DRAM in path granularity. These path requests originate from the trainer_GPU. These path granularity accesses are oblivious because memory accesses to the superblocks are identical to those of PathORAM, and PathORAM access patterns are oblivious. Below we prove that using superblocks does not reveal any additional information over using memory blocks.

Obliviousness of Superblocks: When a superblock is accessed, the path the superblock is assigned to will be fetched. Although multiple data blocks in a path are accessed, every block in the superblock will be assigned to a new uniformly chosen path. Uniformity of new path assignment is proved using total probability[20]. Let N be the total number of paths in the binary tree, p be any path in the binary tree, and b_i be superblock #i. Additionally, let's define $d \bar{\in} b_i$ as a data block d being firstly accessed in superblock #i and $p \in b_i$ as path p assigned to the superblock #i. Then, we can prove the uniformity:

$$\begin{aligned}
&\Pr(NEXT\_PATH(d) = p) \\
&= \sum_{i=1}^{\infty} \Pr(NEXT\_PATH(d) = p | d \bar{\in} b_i) \cdot \Pr(d \bar{\in} b_i) && (1) \\
&= \sum_{i=1}^{\infty} \Pr(p \in b_i | d \bar{\in} b_i) \cdot \Pr(d \bar{\in} b_i) && (2) \\
&= \sum_{i=1}^{\infty} \Pr(p \in b_i) \cdot \Pr(d \bar{\in} b_i) = \sum_{i=1}^{\infty} \frac{1}{N} \cdot \Pr(d \bar{\in} b_i) && (3) \\
&= \frac{1}{N} \cdot \sum_{i=1}^{\infty} \Pr(d \bar{\in} b_i) = \frac{1}{N} \cdot 1 = \frac{1}{N} && (4)
\end{aligned}$$

Equation (1) holds due to total probability. Equation (2) holds due to the definition of the conditional probability in (1). Equation (3) holds because path assignment is independent of data blocks in the superblocks. The rest of the proof just follows arithmetic manipulations.

After path reassignments, the original path the superblock is associated with will the written back using data blocks currently in the stash. Thus, memory access patterns of superblocks are identical to those of the original PathORAM. Adversaries can only observe that random paths are being accessed every time embedding table lookups take place [51]. Since the LAORAM will generate the same memory access behaviors as the original PathORAM and the original PathORAM is oblivious, LAORAM accesses to ORAM server memory are also oblivious.

6.2 Trainer_GPU

The Trainer_GPU acts as a client in our setting, which is considered trusted in ORAM designs. As we mentioned earlier in Section 2, there exist techniques like confidential computing to protect GPUs against a curious OS that are sufficient for our threat model.

6.3 Preprocessor

In LAORAM, the preprocessor reads the training samples and is considered part of the secure client as it can be directly run on the trainer_GPU. When running inside trainer_GPU, adversaries cannot obtain memory access patterns of the preprocessor to GPU VRAM because GPU VRAM is tightly packed via interposers, and hence snooping on the address bus would essentially require breaking the GPU. Moreover, the preprocessor is only responsible for analyzing training samples and creating a window of training batches that are going to be trained later. In the process of generating the look ahead batches, the preprocessor does not need to access any embedding table entries, and hence it does not reveal any sensitive addresses. Thus, accessing the training samples is not a security concern, and the preprocessor is secure.

7 EXPERIMENTAL SETUP

7.1 Methodology & Terminology

We implemented LAORAM as a self-contained memory access engine that has the ORAM server and client architecture functionality implemented across a GPU (ORAM client i.e. trainer_GPU), and CPU with DDR4 memory (server_storage). This infrastructure can be integrated within any embedding table training system. Our system takes a series of memory block addresses generated by a training data preprocessor as inputs and creates superblocks and assigns paths as per the algorithms described earlier. We demonstrate performance improvements of LAORAM with and without fat-tree against the baseline PathORAM.

7.2 Models and Datasets

We use the following datasets and models to quantify the benefits of LAORAM: Permutation Dataset, Gaussian Dataset, Kaggle Dataset, and the XNLI Dataset. The **Permutation dataset** randomly generates an address in the range $0 - N$ where none of the addresses are repeated until all the addresses are accessed at least once. The **Gaussian dataset** generates an address stream sampled from Gaussian distribution. In addition to the two datasets we also use two ML models that train embedding tables and their associated datasets to evaluate LAORAM. We use the DLRM recommendation model [35] that trains large embedding tables using the Criteo AI Labs Ad Kaggle Dataset (Kaggle). The **Kaggle dataset** is released by Criteo AI labs containing real user data and is provided by DLRM as one of its representative recommendation model training datasets. Finally, we use XLM-R [13] a well-known natural language processing model that trains embedding tables using the Cross-Lingual NLI Corpus (XNLI) dataset. The **XNLI dataset** contains multiple data points in multiple languages, and it is used as a key metric for NLP models [27].

The permutation dataset will produce LAORAM's worst-case performance because it will put the biggest pressure on the stash size (proven in the original PathORAM paper [42]). Bigger pressure on stash size results in more dummy reads, leading to the worst-case LAORAM performance. The DLRM and XLM-R models and their associated datasets show the LAORAM performance under a practical application setting.

In our results (Figure 6), we label LAORAM results with fat-tree structure as"Fat", and LAORAM results without fat-tree as "Normal". We also use $/S\#$ notation to indicate the superblock size. For example, "Fat$/S2$" means that the given LAORAM tree uses a fat-tree structure and the superblock size is 2. We used PathORAM [42] as the baseline, which can be treated as an ORAM design with a superblock size of just one. "Fat PathORAM" is PathORAM results with a fat-tree structure. We also use PrORAM as another baseline.

7.3 Embedding Table Configurations

For both Permutation and Gaussian, we have chosen the total number of embedding entries to be 8M and 16M. For the DLRM embedding tables, the largest table from Kaggle has 10131227 entries. Each embedding table entry is 128 bytes. For the XLM-R model, the embedding table has 262144 entries and each entry is $4k$ bytes. The memory required to hold those embedding tables is shown in Table 1. The default bucket size for PathORAM and LAORAM is 4. The reasons ORAM designs suffer from larger memory requirements are the tree structure and the bucket size. As we described earlier, for efficiency of stash management ORAM designs generally use a large bucket size in each tree node, where only a subset of entries within a bucket contains valid data, while the remaining entries contain dummy data. The FAT tree organization uses even bigger bucket sizes at the top of the tree structure. Furthermore, the tree organization itself leads to some fragmentation which further increases the ORAM memory requirements. Note that while one may consider these chosen memory sizes to fit within the high-end GPU HBMs, we chose these sizes to stay within our experimental infrastructure limits on GPU and CPU physical memories. It has already been demonstrated that the size of the embedding tables for recommendation systems in production environments can exceed 100s of GB [33, 54]. With NLP models supporting more languages, we expect the size of embedding tables in NLP to grow as well.

Table 1: Embedding table memory requirement

	Insecure	PathORAM	LAORAM	FAT
8M	1GB	8GB	8GB	10GB
16M	2GB	16GB	16GB	24GB
Kaggle	1.2GB	16GB	16GB	20.3GB
XNLI	1GB	16GB	16GB	20.5GB

System configuration: We conducted our experiments on an experiment platform consisting of an Intel Xeon E-2174G CPU as the ORAM server with 64GB DDR4 memory to store the embedding tables. And a RTX 1080 Ti GPU based system is used as an ORAM client.

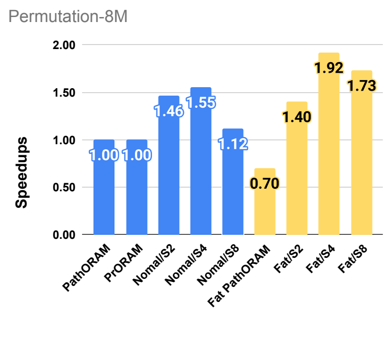
8 EXPERIMENTAL RESULTS

8.1 Comparisons with PrORAM

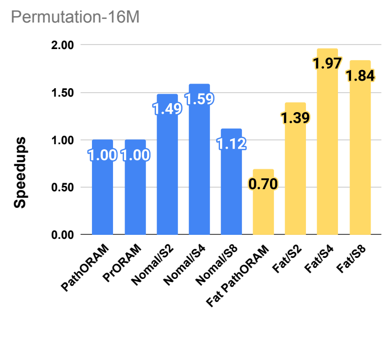
PrORAM [51] is an approach that uses the spatial and temporal locality of data blocks accessed to form superblocks to reduce the number of memory accesses required. It differs from LAORAM in two major ways. First, in PrORAM the superblocks are only formed for data blocks that have neighboring memory addresses. Second, in PrORAM the *past history* of memory accesses is used to find locality and prefetch data. For machine learning systems, the address access patterns to the embedding table are random (See Figure.2 for instance). Hence, intuitively using past history to form superblocks is unlikely to improve performance. The superblock hit rate for PrORAM is extremely low in our experiment. For Kaggle and XNLI datasets, the superblock hit rate is only 6% and 12%, respectively. For the randomized Permutation and Gaussian datasets, the hit rate for PrORAM was close to zero. Thus, PrORAM's performance benefits were correspondingly minimized based on these hit rates (as we quantify in the next section).

8.2 Preprocessing timing

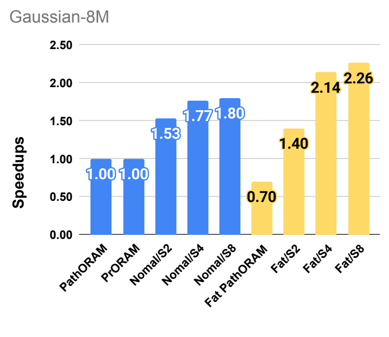
Preprocessing and accessing data are two pipeline stages in the 2-stage LAORAM pipeline. Once the preprocessing for the first several batches is complete, GPU can generate the LAORAM accesses and



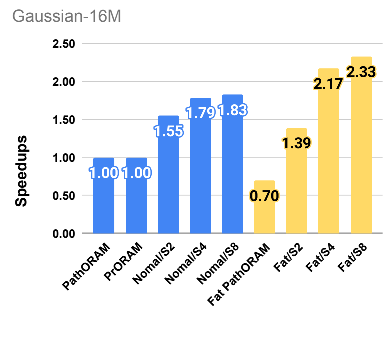
(a) 8M permutation



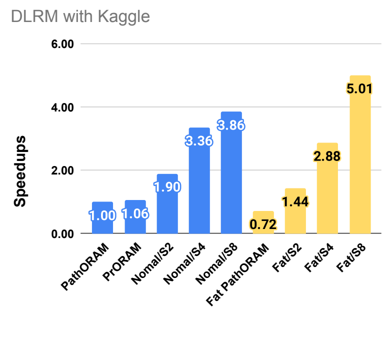
(b) 16M permutation



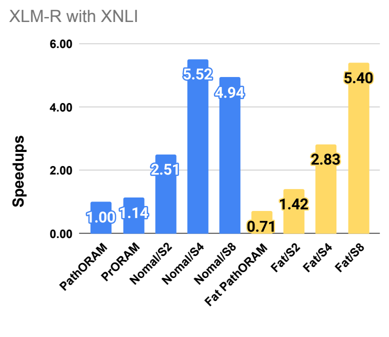
(c) 8M Gaussian



(d) 16M Gaussian



(e) DLRM with Kaggle



(f) XLM-R with XNLI

Figure 6: LAORAM Speedups.

start the training process. The preprocessing can then run ahead of the GPU training process. In our experiments, preprocessing a sample to identify unique indices is orders of magnitude faster than training that sample on a GPU. Thus, the runtime shown in later sections does not include preprocessing since it is not on the critical training path and is small. In our experiments, the initial preprocessing time for setting up the superblocks for **an epoch** was about 10 minutes, which is multiple orders of magnitude less than the training time on GPU.

8.3 LAORAM Access Latency Results

Permutation Dataset: In this section, we evaluate the performance improvement of LAORAM and PrORAM over a PathORAM baseline for the permutation dataset. For these results, we measured the amount of time it took to access a given block from a client's computation viewpoint. The time includes sending the path to be fetched from the trainer_GPU client to the server_storage, the server_storage accessing the tree and fetching all the blocks in the path, and the server_storage sending that data back to the trainer_GPU's stash. As shown in Figure 6a with the 8M permutation dataset, the PrORAM performance improvements (second bar in the graphs) are virtually non-existent, which demonstrates the challenge of using history to form the superblocks. Given the minimal improvements with PrORAM, we focus most of our discussion on LAORAM from here on.

Using LAORAM with a normal tree structure and a superblock size of 2 and 4 gives us a speedup of 1.46X and 1.55X, respectively, over a path ORAM baseline. The speedup shows that forming superblocks using LAORAM's tracking of the upcoming embedding table indices increases performance. Nevertheless, when the superblock size increase to 8, the performance dips to 1.12X because a superblock size of 8 puts greater pressure on the stash and increases the number of dummy reads by 100%, compared to using a superblock size of 4. Although, when using the permutation dataset, $fat/S8$ does not outperform $fat/S4$ due to pressure on the stash, in later results, $fat/S8$ can outperform $fat/S4$ when using real-world data stream.

The "Fat PathORAM" bar in Figure 6a shows the performance impact of the fat-tree structure for PathORAM, and the last three bars in Figure 6a show the performance of LAORAM with a fat-tree. As mentioned before, a fat-tree structure is most useful when there is substantial stash contention and background evictions. With large superblocks, the contention grows, and the fat-tree is designed to handle large superblocks more efficiently. Hence, when using a superblock size of 1 (Fat PathORAM) or 2, the fat-tree organization sees limited if not negative benefits, but with a superblock size of 4 and 8, the fat-tree implementation achieves a substantial performance boost compared to the baseline and outperforms the normal tree implementation. We observe similar behavior for permutation data set the size of 16M as shown in Figure 6b.

ML models: This section discusses speedups for DLRM, XLM-R, and Gaussian datasets. Figure 6c, 6d, 6e and 6f present results for 8M-Gaussian, 16M-Gaussian, DLRM with Kaggle datasets and XLM-R with XNLI datasets. In the original PathORAM paper, the probability of stash overflow is maximized when access patterns contain no duplicate addresses (which is emulated in our Permutation dataset). For DLRM and XLM-R, there are some duplicate addresses within a given window of the access stream.

Note that in Figure 2 even though the address stream is heavily randomized, there is a thin band of repeated accesses at the bottom of the figure. Such repetitive accesses, even when they are a small fraction, do improve the likelihood of placing that block back into the LAORAM path. Hence, the stash growth is reduced. This reduction in turn reduces the number of dummy reads needed later to reduce the stash size. Thus, LAORAM sees higher performance improvements due to this repetition of address accesses.

8.4 End-to-end LAORAM performance benefits

We have also evaluated the ORAM's performance impacts on end-to-end ML training and how LAORAM significantly shortens the gap between private training and training without ORAM in Table 2. We measure the latency in milliseconds per input, and the scale is with respect to the training latency without ORAM. The models we used are XLM-R (L=24,H=1024,A=16) [13], and a large DLRM model. For XLM-R and DLRM, using PathORAM for privacy comes with a cost. When using PathORAM, the latency increases by 8.84x and 56.5x for XLM-R and DLRM, respectively. Compared with XLM-R, which is computation-intensive Transformer-based, DLRM models are based on Multi-Level Perception, which is less computationally intensive. Thus, when using PathORAM, XLM-R sees less training performance impact than DLRM.

On the other hand, when using the best-performing LAORAM configuration, private training latency is reduced to 2.41x and 12.1x w.r.t. training without memory access pattern protection. Compared with PathORAM, the best-performing LAORAM reduces the end-to-end training latency by 3.7**x** and 4.6**x** for XLM-R and DLRM. Especially for the XLM-R model, with a perfect pipeline that overlaps LAORAM management and GPU training, the end-to-end training latency with LAORAM can theoretically be as low as 1.41x. LAORAM greatly reduces the ORAM costs of PathORAM in training embedding tables.

Table 2: End-to-end latency (ms/input); the scale is w.r.t. the training latency w/o ORAM.

Model	w/o ORAM		w/ PathORAM		w/ LAORAM	
XLM-R	Latency	Scale	Latency	Scale	Latency	Scale
	16	1x	154.4	**8.84x**	42.4	**2.41x**
DLRM	Latency	Scale	Latency	Scale	Latency	Scale
	0.61	1x	34.88	**56.5x**	7.52	**12.1x**

8.5 Memory neutral comparison

Using fat-tree increases the storage requirement of the tree structure compared to normal tree with the same number of levels. The increase in storage requirement for the fat-tree compared to the normal binary tree for 8 million and 16 million embedding entries is 25% and 50%, respectively. To discount the benefit of this additional memory, we have also conducted experiments where the normal tree's bucket size is increased such that the total size of the normal tree is at least as big as the fat-tree. In particular, we conducted an experiment where the normal tree uses a bucket size of 6, and a fat-tree's bucket size varies from 5 (leaf node) to a bucket size of 9 (root) for the same number of tree levels. In this organization, the fat-tree uses 16.6% less memory space than the normal tree. While we do not plot the results here due to space limitation, our results show that even with such a memory size handicap, the fat-tree generated 12.4% fewer dummy reads. Increasing the bucket size of all nodes in the tree uniformly does not provide as much performance gain as the fat-tree design because the fat-tree uses memory spaces more effectively by allocating more space for nodes with a higher probability to accommodate data blocks in the stash.

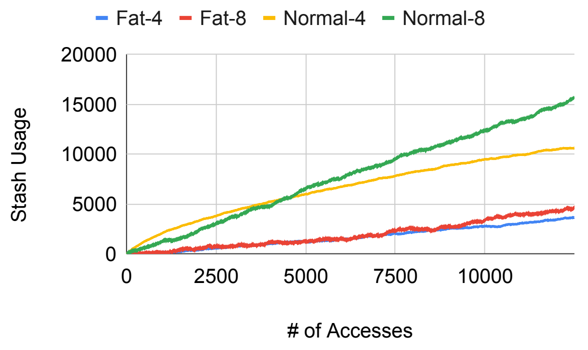


Figure 7: Fat-tree vs normal binary tree. Fat-tree incurs fewer blocks in the stash when the superblock size is large.

8.6 Stash Usage

In Figure 7, we show the stash growth of LAORAM with and without fat-tree using 2 different configurations. The $fat/4$ and $normal/4$ configurations involve a superblock size of 4. We use a bucket size of 4 for the normal tree and 8-to-4 for the fat-tree. Fat-8 and Normal-8 configurations involve a bucket size of 8 and 8-to-16. The graph shows that after around 12500 accesses, the stash for a $normal/4$ grows to around 10600 blocks, while the stash for $fat/4$ grows to about 3600 blocks. After 12500 accesses, $normal/8$ has a stash size of 15500 blocks while $fat/8$ has a stash size of 4700. These results show that for larger block sizes the stash growth for the fat-tree is slower than that of the normal tree.

8.7 Dummy Read Frequency

One way to drain the stash is to perform background evictions whenever the stash usage gets above a particular threshold. In this section, we measure the average number of dummy reads needed per access. In this experiment, dummy reads are triggered whenever the stash size grows above 500 entries, and a series of dummy reads are performed until the stash size reduces to 50 entries.

Table 3 shows the average dummy reads per access for both normal tree structure and fat-tree structure for different superblock sizes for Permutation, Gaussian, and DLRM. The permutation dataset presents the worst pressure on the stash for any PathORAM [42]. For the Permutation dataset, for $normal/4$ and $normal/8$, the average number of dummy reads per access is 0.57 and 1.19. That for fat-tree is 0.14 and 0, 24. When using the fat-tree structure, the total number of dummy reads is reduced by 60% and 75%. A similar phenomenon can be observed in other models. Usage of the fat-tree reduces the number of dummy reads required by nearly 3 times.

Table 3: Average dummy reads per data access

Config	Permutation	Gaussian	Kaggle	XNLI
Fat/S8	0.35	0.24	0.025	0.009
Fat/S4	0.14	0.10	0	0
Normal/S8	1.19	0.65	0.19	0.16
Normal/S4	0.57	0.46	0.053	0

8.8 Traffic Reduction

The advantage of LAORAM is that the design guarantees that all the blocks in a superblock are going to be accessed after a single path is fetched. Comparing fat-tree and normal tree implementation of LAORAM, the bandwidth requirement of fat-tree increases compared to the normal tree by a factor $\frac{3Z+1}{2(Z+1)}$, where Z is the bucket size. Incorporating the increase in bandwidth because of fat-tree and the decrease in bandwidth because of superblocks, we can show that the upper bound on the bandwidth reduction for LAORAM with fat-tree compared to baseline PathORAM is $\frac{2(Z+1)}{3Z+1} * superblockSize$. The upper bound on the bandwidth reduction for LAORAM with normal tree compared to baseline PathORAM is $superblockSize$.

Figure 8 shows the measured bandwidth reductions with DLRM. As shown in the figure, for the normal tree with a superblock size of 2, the bandwidth reduction is 2 times which is equal to our theoretical bound. Theoretical bounds match the empirical data when the number of background evictions is 0. For superblock size 4, the number of background evictions starts affecting the bandwidth reduction, and hence the bandwidth reduction for the normal tree with superblock size 4 is 3.30X which is lower than the theoretical upper bound savings of 4X. The same observations apply to the normal tree with a superblock size of 8.

The bandwidth reduction for fat-tree for smaller superblocks is less than the normal tree; fat-tree is not needed when the number of background evictions is very low. The bandwidth reduction by using $fat/S8$ is more than that of $normal/S8$. Even though theoretically, a single access to fat-tree requires about 50% more bandwidth compared to normal tree, the number of background evictions in fat-tree is very low in comparison to normal tree. We did the same analysis with the permutation dataset, which has higher stash contention. Our results show that fat-tree, in fact, sees a higher traffic reduction factor than the normal tree.

8.9 Comparisons with Ring ORAM

RingORAM [38] is an alternative approach to reduce bandwidth and improve the runtime of PathORAM. RingORAM only reads one block per node read. Theoretically, it reduces the bandwidth by the bucket size. This approach is orthogonal to LAORAM. LAORAM superblocks can be adapted to RingORAM as well. Instead of fetching $n \times log(N)$ data blocks from n paths for every n accesses, with LAORAM, only $[n \times log(N)]/S + S$ blocks from n/S paths need fetching, where S is the superblock size. RingORAM, which

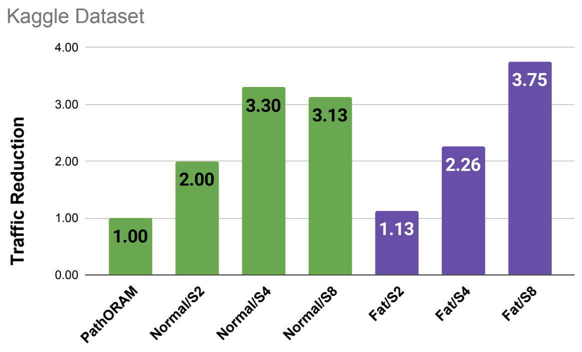


Figure 8: LAORAM memory traffic reduction: DLRM.

uses LAORAM's superblock formation strategy, will face a similar stash overflow conundrum as LAORAM on PathORAM. Our fat-tree structure, a tree structure that better utilizes memory space, can help reduce the number of evictions. Thus, we believe LAORAM speedups, when implemented on RingORAM, are comparable with that of LAORAM on PathORAM.

9 RELATED WORK

Dynamic Superblocks: A further enhancement was proposed in PrORAM [51], which dynamically creates and destroys superblocks using a counter-based history capture scheme. LAORAM relies on future access knowledge as opposed to history-based predictions to provide substantial performance and bandwidth benefits. LAORAM can merge any data blocks into a superblock, while PrORAM can only merge consecutive data blocks.

Comparisons with PrORAM [51] and RingORAM [38] have been discussed in Section 7. In String ORAM[10] a more compact ORAM structure is proposed by leveraging real blocks to obfuscate memory rather than dummy blocks. String ORAM works closely in conjunction with DRAM scheduler to achieve its benefits, while our approach is agnostic to DRAM scheduler knowledge. Range access ORAMs: Lite-rORAM and Hybrid-rORAM are two ORAMs proposed in [11]. They are aimed to increase storage and access efficiencies for multi-range accesses. Our work primarily aims to improve the performance of training recommendation models, where inputs are more randomized. InvisiMem [6] and ObfusMem [8] use the logic layer in Micron's HMC architecture [36] to provide memory obfuscation and reduce the overhead of issuing multiple dummy accesses per real access. They guarantee security by expanding the trusted computing base of ORAM to include the entire memory module with the logic inside it. Because of this assumption that the memory module is secure, both ObfusMem and InvisiMem are only interested in hiding memory access patterns in between the memory and the CPU. This approach requires substantial modification to memory hardware and interface and is built on top of HMC architecture. LAORAM, in comparison, assumes that the memory device i.e. server_storage is insecure and uses traditional DRAM design. Fork Path [53] improves ORAM with a better request scheduler to merge path requests. Similarly, CP-ORAM [47] also designs a special scheduler to accommodate different types of memory requests. Our work does not change any memory request scheduler. Shadow Block[52] improves ORAM accesses by duplicating useful data blocks in dummy blocks, while our work proposes a method to reduce the number of dummy blocks. There are also prior works that incorporate ORAM into other systems, such as [40] and [7]. Our work focuses on providing efficient memory access obfuscation for recommendation models. PageORAM [37] is a recent work that tries to solve the stash growth issue in PathORAM for small bucket sizes. Considering stash growth is an even more major issue for superblocks, the page-based locality of PageORAM can be used to alleviate the stash growth issue in LAORAM.

10 CONCLUSION

In this paper, we present LAORAM, a novel approach to take advantage of the model training where accesses to embedding table memory blocks are known beforehand. LAORAM uses the *future* knowledge to create large superblocks, which are then assigned to the same path in a PathORAM to improve access efficiency. LAORAM presents a novel architecture with three components, the server_storage, the trainer_GPU, and the preprocessor. The preprocessor processes the future access patterns and forms superblocks consisting of data blocks that are accessed together and assigns them to the same path. The LAORAM client then uses this preprocessor-generated metadata to manage the path assignments. To mitigate the stash overflow while using superblocks, a fat-tree structure with a variable bucket size is proposed instead of the usual binary tree with a fixed bucket size. With the improvements mentioned above, LAORAM provides nearly 5X performance improvement compared to PathORAM on two different ML models and different data access patterns.

ACKNOWLEDGMENT

We sincerely thank all the reviewers for their time and constructive comments. This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract Nos. HR001120C0088, NSF award number 2224319, REAL@USC-Meta center, and VMware gift. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] [n. d.]. Hopper Architecture, the Next Generation of Accelerated Computing. https://nvidianews.nvidia.com/news/nvidia-announces-hopper-architecture-the-next-generation-of-accelerated-computing?ncid=em-anno-501708#cid=hpc06_em-anno_en-us. Accessed: 2022-04-13.

[2] [n. d.]. Nearly 70% of US smart speaker owners use Amazon Echo devices. https://techcrunch.com/2020/02/10/nearly-70-of-u-s-smart-speaker-owners-use-amazon-echo-devices/. Accessed: 2022-04-18.

[3] [n. d.]. Recommendations Figuring out how to bring unique joy to each member. https://research.netflix.com/research-area/recommendations. Accessed: 2022-04-18.

[4] [n. d.]. What is Epoch in Machine Learning? https://www.simplilearn.com/tutorials/machine-learning-tutorial/what-is-epoch-in-machine-learning. Accessed: 2023-02-19.

[5] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[6] Shaizeen Aga and Satish Narayanasamy. 2017. InvisiMem: Smart Memory Defenses for Memory Bus Side Channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 94–106.

[7] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious File System for Intel SGX. *Network and Distributed Systems Security (NDSS) Symposium* (2018).

[8] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. ObfusMem: A Low-Overhead Access Obfuscation for Trusted Memories. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 107–119.

[9] Keshav Balasubramanian, Abdulla Alshabanah, Joshua D Choe, and Murali Annavaram. 2021. CDLRM: Look Ahead Caching for Scalable Training of Recommendation Models. In *Proceedings of the 15th ACM Conference on Recommender Systems (RecSys)*.

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

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

[12] Michael Chui, James Manyika, Mehdi Miremadi, Nicolaus Henke, Rita Chung, Pieter Nel, and Sankalp Malhotra. [n. d.]. Notes from the AI frontier: Insights from hundreds of use cases. *McKinsey Global Institute* ([n. d.]).

[13] Alexis Conneau and Kartikay Khandelwal. 2020. Unsupervised Cross-lingual Representation Learning at Scale. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)* (2020).

[14] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016).

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*.

[16] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML)*.

[17] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[18] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC)*.

[19] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996).

[20] John A. Gubner. 2006. *Probability and Random Processes for Electrical and Computer Engineers*. Cambridge University Press.

[21] Hanieh Hashemi, Yongqin Wang, and Murali Annavaram. 2021. Darknight: A data privacy scheme for training and inference of deep neural networks. *Proceedings on the 54th International Symposium on Microarchitecture (MICRO)* (2021).

[22] Hanieh Hashemi, Wenjie Xiong, Liu Ke, Kiwan Maeng, Murali Annavaram, G. Edward Suh, and Hsien-Hsin S. Lee. 2022. Data Leakage via Access Patterns of Sparse Features in Deep Learning-based Recommendation Systems. (2022). arXiv:2212.06264 [cs.CE]

[23] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[24] Rong Jin. 2017. Deep Learning at Alibaba. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*.

[25] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium*.

[26] Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. 2019. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. *arXiv preprint arXiv:1902.04413 [cs.CR]* (2019).

[27] Guillaume Lample and Alexis Conneau. 2019. Cross-lingual Language Model Pretraining. *33rd Conference on Neural Information Processing Systems (NeurIPS)* (2019).

[28] C. E. Leiserson. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.* C-34, 10 (1985).

[29] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[30] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692 [cs.CL]* (2019). arXiv:1907.11692 [cs.CL]

[31] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[32] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*.

[33] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, KR Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. 2021. Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models. *arXiv preprint arXiv:2104.05158 [cs.DC]* (2021).

[34] Krishna Giri Narra, Zhifeng Lin, Yongqin Wang, Keshav Balasubramanian, and Murali Annavaram. 2021. Origami Inference: Private Inference Using Hardware Enclaves. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. https://doi.org/10.1109/CLOUD53861.2021.00021

[35] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv preprint arXiv:1906.00091 [cs.IR]* (2019).

[36] J. Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477494

[37] Rachit Rajat, Yongqin Wang, and Murali Annavaram. 2022. PageORAM: An Efficient DRAM Page Aware ORAM Strategy. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[38] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. 2015. Constants Count: Practical Improvements to Oblivious RAM. In *24th USENIX Security Symposium*.

[39] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2013. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*.

[40] Sajin Sasy, Sergey Gorbunov, and Christopher Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. *Network and Distributed Systems Security (NDSS) Symposium* (2018).

[41] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-Preserving Deep Learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[42] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*.

[43] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. *arXiv preprint arXiv:1806.03287* (2019). arXiv:1806.03287 [stat.ML]

[44] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[45] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2018. SecureNN: Efficient and Private Neural Network Training. Cryptology ePrint Archive, Paper 2018/442.

[46] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-Scale Commodity Embedding for E-Commerce Recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*.

[47] Rujia Wang, Youtao Zhang, and Jun Yang. 2017. Cooperative Path-ORAM for Effective Memory Bandwidth Sharing in Server Settings. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[48] Yongqin Wang, Rachit Rajat, and Murali Annavaram. 2022. MPC-Pipe: an Efficient Pipeline Scheme for Secure Multi-party Machine Learning Inference. arXiv:2209.13643 [cs.CR]

[49] Yongqin Wang, G. Edward Suh, Wenjie Xiong, Benjamin Lefaudeux, Brian Knott, Murali Annavaram, and Hsien-Hsin S. Lee. 2022. Characterization of MPC-based Private Inferences for Transformer-based Models. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[50] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*.

[51] Xiangyao Yu, Syed Kamran Haider, Ling Ren, Christopher Fletcher, Albert Kwon, Marten van Dijk, and Srinivas Devadas. 2015. PrORAM: Dynamic prefetcher for Oblivious RAM. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[52] Xian Zhang, Guangyu Sun, Peichen Xie, Chao Zhang, Yannan Liu, Lingxiao Wei, Qiang Xu, and Chun Jason Xue. 2018. Shadow Block: Accelerating ORAM Accesses with Data Duplication. *International Symposium on Microarchitecture* (2018), 961–973.

[53] Xian Zhang, Guangyu Sun, Chao Zhang, Weiqi Zhang, Yun Liang, Tao Wang, Yiran Chen, and Jia Di. 2015. Fork Path: Improving Efficiency of ORAM by Removing Redundant Memory Accesses. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*.

[54] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. 2019. AIBox: CTR Prediction Model Training on a Single Node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management (CIKM)*.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact consists of two parts: 1) statistic generation and 2) timing measurements. The first part includes the C++ code that reads pre-processed trace files to measure the number of accesses to the ORAM tree, and the number of dummy reads in an epoch of the XNLI dataset. The second part is Python and C++ code that measures the runtime to access ORAM once, and the runtime of a single dummy reads. Both results from those two parts generate the key results in Figure 6f.

A.2 Artifact check-list (meta-information)

- **Algorithm:** New PathORAM access strategy.
- **Compilation:** GNU C++ compiler and NVCC CUDA compiler.
- **Binary:** oramTest.
- **Model:** Embedding Tables.
- **Data set:** XNLI.
- **Hardware:** Intel(R) Xeon(R) Gold 5220R and an Nvidia RTX 1080 Ti.
- **Metrics:** Runtime Comparison.

A.3 Description

A.3.1 How to access. Our source code and benchmarks would be provided on request.

A.3.2 Hardware dependencies. We require an Nvidia GPU that supports CUDA to get the timing measurements.

A.3.3 Software dependencies.

A.3.4 Data sets. The pre-processed trace files for XNLI are provided in the link.

A.3.5 Models. A embedding table whose size is 262144×1024.

A.4 Installation

Compiling the statistic generation code:

```
g++ statORAM/oram2.cpp -o testORAM
```

Compiling the timing generation code:

```
make -C pretch-oram/
```

Install Python dependencies: Please firstly change the python3.x-venv in the file install_python.sh to the corresponding Python version of your machine.

```
./install_python.sh
mkdir [a random empty called garbage]
```

The "mkdir" step is very important and cannot be omitted.

Download pre-processed trace files: There are 4 files in the download link provided link. The number in the file name indicates the batch size, and files with "init" in the name are the metadata for position maps.

A.5 Experiment workflow

A.5.1 XNLI Baseline PathORAM. **Statistics:** Because when using PathORAM, there is no dummy reads required to empty the stash, there is no need to run statistical code for the PathORAM. The total number of accesses to the ORAM is 3149694, and there are no dummy reads.

Timing:

source oram_sim/bin/activate

export TORCH_EXTENSIONS_DIR=[absolute path to the garbage folder]

python3 -um python.prefetch.scripts.init_test –size=262144 –super=1

The last command will print an ORAM access time. Multiply the total number of accesses to the ORAM, and the ORAM access time will produce the baseline runtime for the epoch.

A.5.2 XNLI LAORAM Fat/s4. **Statistics:** Run command below will produce the total number of dummy reads and the total number of accesses to the ORAM.

./testORAM 4 initXNLI4epoch0.txt oXNLI4epoch0.txt 262144 1 500

Timing: Before running code, please modify the global variable "normalTree" in the file "pretch-oram/pathoramClient.cc" to "false". Then, the four commands below will produce the ORAM access time and dummy read time.

make -C pretch-oram/

source oram_sim/bin/activate

export TORCH_EXTENSIONS_DIR=[absolute path to the garbage folder]

python3 -um python.prefetch.scripts.init_test –size=262144 –super=1

ORAM access time $*$ the number of ORAM accesses + dummy read time $*$ the number of dummy reads is the total runtime for LAORAM Fat/s4.

A.5.3 XNLI LAORAM Fat/s8. **Statistics:** Run command below will produce the total number of dummy reads and the total number of accesses to the ORAM.

./testORAM 8 initXNLI8epoch0.txt oXNLI8epoch0.txt 262144 1 500

Timing: Before running code, please modify the global variable "normalTree" in the file "pretch-oram/pathoramClient.cc" to "false". Then, the four commands below will produce the ORAM access time and dummy read time.

make -C pretch-oram/

source oram_sim/bin/activate

export TORCH_EXTENSIONS_DIR=[absolute path to the garbage folder]

python3 -um python.prefetch.scripts.init_test –size=262144 –super=1

ORAM access time $*$ the number of ORAM accesses + dummy read time $*$ the number of dummy reads is the total runtime for LAORAM Fat/s8.

A.5.4 Gaussian Baseline PathORAM. **Statistics:** Because when using PathORAM, there is no dummy reads required to empty the stash, there is no need to run statistical code for the PathORAM. The total number of accesses to the ORAM is 16312910, and there are no dummy reads. Before running the code, please modify the global variable "**normalTree**" in the file "pretch-oram/pathoramClient.cc" to "true". Also, please change line 21 from "**embedding_dims=1024**" to "**embedding_dims=16**" in file python/prefetch/scripts/init_test.py.

Then, the four commands below will produce the ORAM access time and dummy read time.

Timing: make -C pretch-oram/
source oram_sim/bin/activate
export TORCH_EXTENSIONS_DIR=[absolute path to the garbage folder]
python3 -um python.prefetch.scripts.init_test –size=8156454 –super=1

The last command will print an ORAM access time. Multiply the total number of accesses to the ORAM, and the ORAM access time will produce the baseline runtime for the epoch.

A.5.5 Gaussian LAORAM Fat/s4. **Statistics:** Run command below will produce the total number of dummy reads and the total number of accesses to the ORAM.

./testORAM 4 initNorm4epoch0.txt oNorm4epoch0.txt 8156454 1 500

Timing: Before running code, please modify the global variable "**normalTree**" in the file "pretch-oram/pathoramClient.cc" to "false". Then, the four commands below will produce the ORAM access time and dummy read time.

make -C pretch-oram/
source oram_sim/bin/activate
export TORCH_EXTENSIONS_DIR=[absolute path to the garbage folder]
python3 -um python.prefetch.scripts.init_test –size=8156454 –super=1

ORAM access time $*$ the number of ORAM accesses + dummy read time $*$ the number of dummy reads is the total runtime for LAORAM Fat/s4.

A.5.6 Gaussian LAORAM Fat/s8. **Statistics:** Run command below will produce the total number of dummy reads and the total number of accesses to the ORAM.

./testORAM 8 initNorm8epoch0.txt oNorm8epoch0.txt 8156454 1 500

Timing: Before running code, please modify the global variable "normalTree" in the file "pretch-oram/pathoramClient.cc" to "false". Then, the four commands below will produce the ORAM access time and dummy read time.

make -C pretch-oram/
source oram_sim/bin/activate
export TORCH_EXTENSIONS_DIR=[absolute path to the garbage folder]
python3 -um python.prefetch.scripts.init_test –size=8156454 –super=1

ORAM access time $*$ the number of ORAM accesses + dummy read time $*$ the number of dummy reads is the total runtime for LAORAM Fat/s8.

A.6 Evaluation and expected results

A.6.1 Number of accesses. The number of dummy reads for LAORAM fat/s4 is expected to be 0 or very small. The number of dummy reads for LAORAM fat/s8 is expected to be around 27825. The number of ORAM accesses for fat/s8 should be half of that of fat/s4. The final speedup for fat/s4 should be around 3x, and that of fat/s8 should be around 5.3x.

A.6.2 Timing. ORAM access time and dummy read time will vary depending on the hardware. However, dummy reads should take less time than ORAM accesses because there are fewer workloads for dummy reads.