Efficient Memory Side-Channel Protection for Embedding Generation in Machine Learning

Muhammad Umar[1*], Akhilesh Parag Marathe[3*], Monami Dutta Gupta[3], Shubham Jogprakash Ghosh[3], G. Edward Suh[1,2], Wenjie Xiong[3]
[1]Cornell University, [2]NVIDIA, [3]Virginia Tech
mu94@cornell.edu, esuh@nvidia.com, {akhimarathe,monami,shubhamjghosh,wenjiex}@vt.edu

***Abstract*—Modern machine learning (ML) models need to process both continuous and categorical/discrete feature values, e.g., deep learning recommendation models (DLRMs) rely on users' categorical features to make recommendations, and large language models (LLMs) take discrete words/tokens as input. ML models process such discrete features by converting them to numerical vectors called *embeddings*. Unfortunately, embedding table lookups are vulnerable to side-channel attacks, as table indices leak input feature values. Due to the size of the embedding tables, using conventional oblivious computing techniques such as ORAM to protect memory access patterns to the tables incur significant overhead. In this paper, we propose to use a different technique, Deep Hash Embedding (DHE), to secure embedding table accesses, even though it is not commonly used today due to its compute-intensive nature. We investigate three embedding generation methods with side-channel protection: linear scan of the embedding table, embedding table protected by ORAM, and DHE. Our experiments on DLRMs and LLMs show that DHE or a hybrid scheme combining DHE and linear scan can significantly improve both performance and memory footprint compared to the conventional ORAM protection. For DLRM on Criteo datasets, our hybrid scheme improves performance by about 4× for large embedding tables, and up to 3.08× end-to-end over the optimized ORAM baseline without any loss in accuracy, while reducing the model memory footprint by up to 1116×. For a GPT-2 LLM, using DHE speeds up the prompt prefill by up to 1.32× and decoding by up to 1.07× over ORAM, depending on the batch size, with comparable output quality.**

I. Introduction

Machine learning (ML) models are increasingly deployed in a wide range of tasks. However, the data required to train and run ML models can often be sensitive and needs protection. For example, deep-learning based recommendation models (DLRMs) [80] are widely used for personalized recommending and ranking items in e-commerce [8], [133], [134], social media [45], entertainment [129], and search [15]. DLRM leverages data specific to individual users, containing the users' sensitive information. Similarly, large language models (LLMs) [10], [90], [109] may take potentially sensitive queries. Moreover, ML tasks are both compute and memory intensive, and ML training and inference tasks are often deployed on the cloud, where there are concerns about cyber attacks, malicious service providers, co-located attackers, etc. As a result, privacy-preserving machine learning (PPML) techniques have been proposed to protect the ML workloads, including cryptography-based schemes [65], [93], [96], [115] and hardware-based trusted execution environments (TEEs) [7], [17], [18], [53]. However, the memory access pattern of the execution is usually not fully protected in today's computing systems, leaving the door open for side-channel attacks.

Meanwhile, ML models have evolved to deal with more data types. While traditional ML models mainly use continuous data such as pixel values in images, ML models can also use non-numerical features that are discrete or categorical, e.g., the tokens in LLMs and the categorical user features in DLRM. For ML models to handle discrete features in continuous vector space, the categorical features need to be converted to *embeddings*, which are high-dimension vectors capturing the semantics of features. Typically, *embedding tables* are used to generate (i.e., fetch) embeddings for discrete features, via simple lookups.

Unfortunately, the use of embedding tables introduces a memory side-channel vulnerability that can leak the table index of a query, which represents the value of the feature in the request. For example, in a language model, if a token ID is leaked, then the word/subword in an input is leaked. For DLRM, private user information such as age group, gender, purchase history may be leaked by tracking the access pattern to embedding tables [47], [91]. The whitepapers from TEE hardware manufacturers [56] indicate the software design should be hardened against side-channels when sensitive data is involved. While oblivious random access machine (ORAM) and private information retrieval (PIR) are cryptographic algorithms that can hide access patterns in databases, they often incur significant performance and memory overheads [92], [98]. For instance, we evaluate the ORAM overhead on the embedding lookup latency to be an average of 40× for sparse features in DLRM, and 200× for LLM *prefill* with 256 tokens.

In this paper, we aim to enable efficient protection for embedding generation for categorical/discrete features in ML models, using DLRM and LLM as case studies. At the model level, we go beyond embedding tables and study different methods for embedding generation considering both security and performance. In particular, recent research proposed Deep Hash Embedding (DHE) [61], which uses hash functions and fully-connected layers to generate (i.e. compute) embeddings at runtime. Without considering security, DHE is compute-intensive, has longer latency, and thus is only efficient when

*Equal contribution.

DOI: 10.1109/HPCA61900.2025.00041

memory capacity is the main system bottleneck [50]. But interestingly, DHE directly performs computation on categorical feature IDs, and its memory access pattern is independent of a feature value. Thus, DHE is secure against side-channel attacks on memory access patterns. Our study shows that DHE or a hybrid scheme based on DHE can provide better performance and a much smaller memory footprint than traditional embedding lookup with ORAM protection.

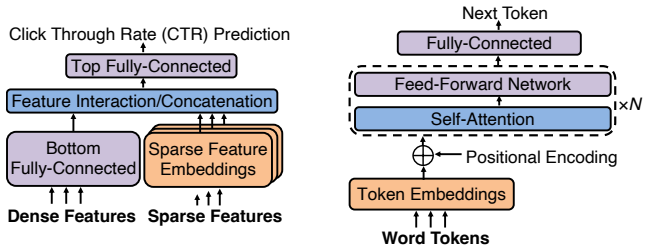
We summarize our contributions as follows:

- We demonstrate security vulnerabilities when executing an ML model with embedding tables on today's commercial confidential computing platforms, by showing a side-channel attack in the Intel SGX TEE.
- We study different embedding generation methods to prevent side-channel attacks on embedding table lookups. To the best of our knowledge, this paper represents the *first* study on secure embedding generation methods combining both the model architecture and oblivious computing technologies for confidential computing.
- We analyze the end-to-end security of DLRMs and LLMs in terms of memory and control flow side-channels. The protection for embedding generation poses the main challenge whereas other layers are secure or can be protected with negligible performance overhead.
- For DLRM, based on our characterization of real platforms, we propose a hybrid embedding generation scheme. We observe that a combination of linear scan and DHE shows the best performance in practice. For a single DLRM inference, the hybrid scheme shows a performance improvement of up to 3.08× over a baseline ORAM scheme, with no accuracy loss, while reducing the model memory footprint by up to 1116×.
- For LLMs, we propose to use DHE for batched secure token embedding generation. This paper is the *first* study that evaluates DHE for LLMs. Applying DHE to a GPT-2 model and finetuning the model gives perplexity comparable to the original model, with up to 1.32× speed-up for prefill and up to 1.07× speed-up for decoding with batching compared to ORAM, and an overhead of only 2–5% over the non-secure implementation with no memory side-channel protection.
- The code of our implementation is publicly available at https://github.com/bearhw/SecEmb_DHE.

II. Background

A. Neural Networks and Embedding Generation

Deep neural network based ML models use a network of interconnected nodes (or neurons) and non-linear activations to process data. Such ML models can effectively process continuous numerical data (e.g., pixel values) as inputs. These numerical inputs are often called dense features. However in certain tasks, ML models also need to process non-numerical features that are discrete or categorical, e.g., the words in LLMs and the categorical user features in DLRM. These categorical features are also called sparse features. *Embedding vectors* are introduced to represent categorical features as numerical values. Below are two popular ML sub-fields where embeddings play an important role.



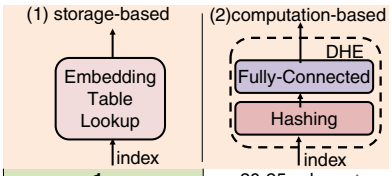
(a) DLRM (b) LLM

Fig. 1: ML models with embedding layers.

Deep Learning Recommendation Model (DLRM): DLRMs [80] take both dense (i.e., numerical) and sparse (i.e., categorical) features as input to predict a match between a user and an item, such as click-through rate (CTR), as shown in Figure 1(a). In DLRMs, fully-connected (FC) neural network layers are used. Sparse (categorical) features cannot be represented by continuous numbers, e.g., the name of the items viewed by a user. Sparse features have to be transformed into *embedding vectors*. Then, the embedding vectors and the processed dense features are combined through feature interaction and further processed by the top FC to give a recommendation probability. The need to process large sparse features distinguishes DLRMs from other ML models. In DLRM, a *batch* of embedding generation from a single feature/table spans lookup IDs from multiple user requests.

In DLRMs, there are typically tens to hundreds of sparse features. Some of the features can contain tens of millions of entries (the embedding table size). The embedding dimension for each entry is typically 16, 32, or 64 [43], [45], [50].

Large Language Models (LLM): Transformer-based LLMs [10], [90], [109] are the most popular models for natural language processing (NLP) tasks such as semantic search, or text completion/generation. The general Transformer architecture takes an input text prompt, which is first converted into discrete tokens (subwords) by a tokenizer. These categorical tokens need to be converted into individual *token embeddings* to be processed further, as shown in Figure 1(b). The token embeddings are obtained by looking up a token embedding table using the token IDs as lookup indices. Then, a positional encoding is added to token embeddings, and the sum is processed through many attention and feed-forward layers. The result can be either used as the prompt's semantic representation or in the generative version of the Transformer, the result is transformed (using an FC layer *head*) into logits for the output token. The new token is then fed back as appended input to generate the next output token, generating text autoregressively token-by-token until the end of the sentence token is reached. In this study, we focus on generative Transformers. During generation, there are two stages as described above: *prefill* (input prompt processing) and the subsequent *decoding* (autoregressive token generation). The prefill stage accesses possibly multiple token embeddings for the entire



		(1) storage-based	(2) computation-based
Non-secure	latency	**1x**	~20-25x slower↑
	mem.	**1x**	~0.01x smaller↓↓↓
Secure	latency	~30x-50x slower↑↑	**~20-25x**
	mem.	~3x larger↑↑	**~0.01x**

Fig. 2: Comparison of embedding generation methods in ML models. *mem.* refers to memory footprint. The number shows the normalized performance in DLRM (batch size = 32). The smaller the better.

input prompt, whereas decoding accesses only one at a time (for a single inference); these result in different embedding layer *batch* sizes. The output FC layer head typically shares weights with the token embedding table [85] for better quality.

LLMs only have one index-based embedding table (i.e., for tokens), whereas DLRMs have multiple tables of different sizes for sparse features. Across different models, the LLM token embedding table typically has a few tens of thousands of entries [10], [25], [42], [60], [108]. The embedding dimensions are typically in the range of 768–8192, much larger than those in DLRM.

Taxonomy of Embedding Generation Methods: Today, embedding generation methods usually rely on a table lookup. Such a *storage-based method* stores a trained mapping between a categorical feature value and the corresponding embedding in an embedding table, and the embedding generation can be done by a table lookup using an index value; see Figure 2 (1). For DLRMs, the embedding tables in a model can take *gigabytes or even terabytes* of memory [50].

More recently, researchers have explored embedding generation without looking-up huge embedding tables, as large tables can make the system inefficient due to the high memory usage. The *computation-based method* instead uses compute-heavy *encoder-decoder* stacks to generate embedding vectors using the categorical feature value [50], [62]. In Deep Hash Embedding (DHE), as shown in Figure 2 (2), a parametric hash function is used to *encode* the input ID into multiple values, which are then *decoded* into an embedding via FC layers. DHE was previously studied in DLRMs [50], [62], because embedding generation contributes the majority of the inference time and model size in DLRM. DHE in DLRM provides a design option to trade-off latency and memory footprint. However, DHE is not widely used in practice due to its computation overhead. Also, DHE has not been studied for other models such as LLM, because the embedding table is typically not the bottleneck in models like LLMs.

The table in Figure 2 shows a comparison between the storage-based method and the computation-based method in terms of latency and memory footprint. Without considering security, embedding generation in practice today uses the storage method [87], because memory lookup is much faster than computing an embedding at run time, although DHE takes a smaller memory footprint. However, from the security perspective, memory lookup is vulnerable to side-channel attacks, which are expensive to protect using traditional protection methods such as ORAM. Under security requirements, in this paper, we show that DHE offers competitive latency and a much smaller memory footprint than traditional protection techniques.

CPU vs. GPU: In this study, we focus on CPU platforms. Because large embedding tables in DLRM often cannot fit in the limited GPU memory, embedding layers in DLRM typically rely on CPUs. Recent studies show that the use of CPU DRAMs for large embedding tables in DLRM instead of GPU memory can improve performance and reduce the cost [57], [63], [128]. Even though LLMs are typically trained and served on GPUs today, there are recent efforts to run model inference on CPUs as well, given the need to run LLM in different use-case scenarios and the low cost of CPU platforms. LLMs on CPUs are becoming more feasible by leveraging techniques such as quantization and SIMD vector units across multi-core processors [30], [55].

B. Confidentiality Protection

Confidential Computing uses hardware-based trusted execution environments (TEEs) [6], [9], [12], [16], [19], [31], [35], [68], [70], [77], [102]–[104], [118], [123] to provide confidentiality and integrity protection in the cloud even with an untrusted OS or from physical attacks. Compared with other protection using cryptography such as homomorphic encryption (HE) [93], or secure multi-party computation (MPC) [65], [96], [115], hardware-based protection has a smaller overhead and is more practical to deploy real large-scale applications.

Intel's Software Guard Extension (SGX) [77], the TEE we use in this work, represents one of the state-of-the-art TEE designs. SGX establishes a secure environment called an enclave with a special protected memory region called the enclave page cache (EPC). SGX protects EPC from all non-enclave memory accesses (both reads and writes) from potential *software attackers*, including accesses from the OS kernel, the hypervisor, and peripherals, via built-in hardware isolation mechanisms. To deal with *hardware attackers* who can conduct cold-boot attacks or bus probing, the confidentiality of the protected memory region is maintained via data encryption in the main memory. There are two generations of memory encryption in SGX. The now obsolete Intel Client SGX edition provides a maximum of 256 MB protected memory, due to the costly Merkle-tree based protection for replay attacks [41]. To scale the size of the protected memory to the terabyte-range, the recent server TEEs such as Scalable SGX, AMD SEV [7], and Intel TDX [54] use AES-XTS mode for memory encryption [29], [52], which however, does not protect against replay attacks. Thanks to the large size of the protected memory, we can put the entire DLRM and LLM inside the secure enclave memory.

In today's TEEs, side-channel attacks are not prevented by hardware and are considered as the responsibility of software. Software needs to be written carefully to avoid side-channel

leakage [56]. Moreover, recent studies show that side-channel attacks are practical on public clouds [131], [132].

Data-Oblivious Programs are those whose control/data flow does not depend on the input data [92], [98]. Data-oblivious programs avoid information leakage through side-channels such as memory access patterns. Algorithms with data-dependent memory accesses can be made oblivious by rewriting them in a deterministic fashion e.g., eliminating data-dependent branches. Moreover, algorithms can also hide their memory access patterns via access pattern obfuscation. A simple example is that on each memory access, the algorithm touches all memory blocks in memory through a *linear scan*. More interestingly, Oblivious RAM (ORAM) [38] hides an algorithm's data access pattern via shuffling and re-encrypting the data on each access, while allowing only a negligible probability of learning anything useful about the original access pattern from the obfuscated pattern. Many types of ORAM schemes have been proposed [49], [101], [107], [113]. In confidential computing platforms, ORAM can be available either as a hardware component e.g., inside the memory controller or implemented in software. ORAM is not yet available in today's TEE hardware. As a result, software protection is necessary for full protection when memory access patterns may contain confidential information.

C. Side-channel Attacks on ML Models

Previous studies have shown that side-channel attacks can be applied to ML models. For example, memory access patterns during ML inference can be used to infer the model architecture [51], [122] and model parameter values when the memory accesses are optimized with dynamic pruning [51]. In some implementations, the memory access patterns may also leak output labels [5], [106]. These studies demonstrated the potential risk of side-channel attacks on ML and led to follow-up studies on potential protection techniques including oblivious computing [14], [74], [81], [106]. However, the previous studies on ML side-channels largely focused on the ML models without embedding tables, such as FC or convolutional neural networks. In this work, we experimentally demonstrate that microarchitectural side-channel attacks indeed pose a realistic threat for embedding table accesses in ML models, and develop an effective oblivious computing technique for embedding generation.

III. Threat Model

This paper aims to enable efficient protection in machine learning (ML) models from memory access and control flow side-channel attacks, especially the embedding generation layer. In today's table-based embedding generation, memory access patterns leak sparse feature values that are used as table indices directly. Our goal is to hide the sparse feature values (embedding table indices). The proposed secure embedding generation enables protecting the confidentiality of input values (user and item features, or language tokens), intermediate states, and outputs. We assume that the frequency of inference requests, the number of sparse features accessed, or the model architecture do not need to be hidden.

We assume that the ML workload runs inside a protected execution environment, where attackers cannot access the memory space of the victim ML workload directly, e.g., a hardware-based TEE, a virtual machine (VM), or a software container. However, information leakage is possible due to the side-channel in the *memory access pattern or control flow*. Memory side-channel attacks have been demonstrated to be practical and powerful attacks on the cloud in CPU [73], [86], [131], [132], on-chip and off-chip GPUs and accelerators [27], [28], [79], [126]. We do not consider physical side-channel attacks such as the power and electromagnetic side-channels. We also assume protection against floating-point side-channels [66] as orthogonal to our work.

As a concrete example, this paper implements and evaluates the side-channel protection for DLRM/LLM models running inside an Intel SGX enclave. In LLMs, we assume that the tokenization's encoding and decoding processes between natural language tokens and their token IDs happen on a trusted local device and not in an untrusted cloud. The tokenizer is typically open-sourced, even in proprietary models [39], [82], to enable users to calculate the token count for their prompts and estimate the resulting API usage cost.

A. Information Leakage through Embedding Table Accesses.

(1) Sensitive information that can be inferred from the indices. Embedding table accesses or indices contain a user's private data and accesses to sparse feature tables in DLRM may reveal sensitive user data [47], [91], such as their gender, education level, living city, etc. For instance, the Taobao Ads dataset [105] has many tables describing private user features; the gender table with 2 entries has indices 0 and 1 corresponding to *Male* and *Female* respectively. In LLMs, the user input prompt is converted to token IDs which are used to index the token embedding table, and hence can directly reveal which tokens the user prompt contains. The tokens can be used to reverse-engineer sensitive information in the prompt or the generated text.

This type of memory access pattern attack leaks sensitive input values more directly than other attacks on ML models, such as those involving building statistical models on data-dependent access patterns (e.g., ReLU activation) of CNN models to guess the output class for an input image [106].

(2) Leaking indices through side-channels. Memory access patterns can be leaked via a number of different channels, including cache [73], [131], [132], page faults [121], memory bus [24], and memory row buffer [84]. In the practical datasets we studied, an embedding table entry is always bigger than one cache line, and cache line granularity attack is accurate enough to leak the indices.

Here, we demonstrate how the attacker can obtain the target/victim index to an embedding table for a baseline DLRM implementation on Intel SGX via a side-channel attack in the last-level cache. We implemented an embedding layer in C++ and ran it in the Intel SGX enclave on an Intel Ice Lake Xeon CPU. Our prototype assumes the attacker can pinpoint the starting of embedding access by using the starting of the enclave *ecall*, given that the embedding is the first layer in the

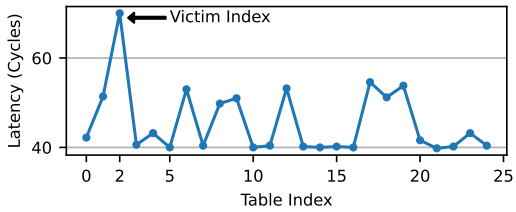


Fig. 3: Access latency of the eviction set observed by the attacker. Here, the actual victim index is 2, and the attacker observes a longer latency for eviction set 2. In DLRM, the index may represent the item ID of a user's recent purchase.

model. The table we tested has 256 entries with an embedding dimension of 64, which represents a small-medium table in the Kaggle or Terabyte datasets [21], [59]. While the enclave program is running on one of the cores on the server processor, the attacker can run processes on a different core on the same processor, allowing the attacker to make memory accesses to the shared last-level cache (LLC) and observe the memory access latency. The attack has two phases. Phase (i) is creating the eviction set for each index. We let the attacker be one of the users of DLRM and query indices in the embedding table to help create the eviction set for each index. We adopt the PRIME+SCOPE attack [86] to the Intel SGX enclave program. In this demonstration, we use the physical address of the embedding table to accelerate the construction of the eviction set. This assumption is also used in other cache side-channel attacks on Intel SGX where a malicious OS can find out the physical address [112]. Phase (ii) is to measure the time for accessing the eviction set. We iterate through each cache set to infer the victim index. Figure 3 shows the results of our attack, each data point in the figure is an average of 10 measurements. The attacker can learn the victim index through timing measurements. Although we have a larger table, we only prime 25 cache sets to demonstrate the feasibility of such an attack in the cache.

In addition to cache side-channel attacks, the table lookup index can be directly leaked if the attacker can see the memory traffic on the memory bus. The indices can also be leaked in attacks using page faults [121]. The OS can reset the present bits of embedding table memory so that every table lookup triggers a page fault. Then, the OS can observe the page-level access patterns. The enclave program's access pattern can also be leaked through the timing of TLB, DRAM row buffer [84], and a combination of cache and DRAM channels [112]. Each of the attacks will leak the index with different granularity. A combination of the attacks can *scale* the attack to leak the exact indices of a large table. For example, page fault or DRAM row buffer can leak coarse-grand address, and cache side-channel can leak the indices within page or DRAM row granularity.

IV. SECURE EMBEDDING GENERATION

In this section, we first study techniques for secure embedding generation against side-channel attacks, either by hiding the index used to look up a table, or by deterministically computing the embedding without a table lookup. Then, based on their performance, we analyze how to make best use of these techniques (possibly in a hybrid manner) for secure and efficient DLRM and LLM designs. While we target secure inference, our analysis can be adapted to training as well.

A. Techniques for Secure Embedding Generation

*1) Embedding Tables protected by **Linear Scan**:* When using the original table representation of embeddings, i.e., a *storage* method, we can use a linear scan to hide the lookup index. Linear scan is a naïve algorithm that scans the entire table for looking up a single index. It has the complexity of $O(n)$ where n is the number of table rows, which is clearly prohibitive for large tables.

*2) Embedding Tables protected by **Tree-based ORAM**:* When using the table *storage* representation, we can apply Oblivious RAM (ORAM) for better scalability in large tables. The standard tree-based ORAM approaches cleverly organize the protected table memory into a tree structure. We consider two tree-based ORAM approaches in this paper:

Path ORAM [101]: A simple ORAM scheme that organizes data into a tree of node buckets, each having Z data blocks. In addition to the tree, Path ORAM has two more data structures: the position map (pos-map) that associates each data block to a tree leaf, and a stash to store data blocks locally. Path ORAM maintains the invariant that data either resides in the stash or on a path on the tree to its assigned leaf. On a data access, the pos-map is looked up for the corresponding leaf, and that path from the tree is fetched to the stash, and the desired block is returned. The block is assigned a new random leaf in the pos-map. The fetched path is then written back to memory by pushing valid blocks in the stash to the path as deep as possible.

Circuit ORAM [113]: It is also a tree-based ORAM, similar to Path ORAM up to fetching the read path. It differs from Path ORAM in how it inserts blocks from the fetched path into the stash, and how it evicts blocks from stash. Unlike Path ORAM, it does not insert the entire path into the stash, but only the relevant block. Moreover, for eviction, it does not iterate over the entire stash repeatedly for each bucket in a path to build the evicted path (for hiding the memory access pattern in oblivious computing in a software-based ORAM controller), but only runs through the stash once by preparing metadata in advance to select appropriate blocks to push into the evicted path. For eviction, Circuit ORAM incurs additional bandwith overhead by fetching two more paths. However, Circuit ORAM works with a much smaller stash than Path ORAM (15× smaller in our setup). As such, the number of instructions/iterations in Circuit ORAM are reduced, and it is better suited to oblivious computing. We refer readers to literature for details on Circuit ORAM [49], [98], [113].

For embedding tables, a per-table ORAM is instantiated. However, embedding tables in DLRM can grow to millions of entries, and using tree-based ORAM techniques for such tables is still expensive, since they lead to a blow-up in bandwidth, and require multiple memory accesses for a single embedding

Algorithm 1: Deep Hash Embedding (DHE) (see [61])

Input: categorical feature value x
Output: an embedding vector
Step 1: encode x into k different values by using k universal hashing functions [11] i.e., calculate $y_k = (((a_k x + b_k) \mod p_k) \mod m)$ where a, b, m, p are integers and m gives the hash bucket size (we have $m = 10^6$)
Step 2: uniformly transform the integer values y_k into real values in the range [–1, 1]
Step 3: decode/transform the k-long vector y using FC layers to give an embedding vector

lookup. The tree organization in ORAM requires additional *dummy* blocks and incurs a large memory space overhead.

3) ***Deep Hash Embedding (DHE):*** DHE [61] is an alternative *computation*-based method for embedding generation, as opposed to *storage*-based methods. DHE first encodes the input feature or index using a series of hash functions, and then decodes the resulting values by passing them together through a fully-connected (FC) network to give an embedding vector. Whereas DHE was originally proposed as a compute-based alternative to lookup-based table storage as a way to reduce the memory footprint and bandwidth usage, we propose to use it as a secure embedding generation technique since its memory access patterns are deterministic and do not depend on the input categorical feature value. Algorithm 1 summarizes the DHE-based embedding calculation (details in [61]). The number of hash functions k and the proportionately large FC sizes in the DHE encoding stage greatly affect the quality of the generated embedding and hence the accuracy of the end-to-end model. In this paper, we setup DHE models to match the baseline table representation's accuracy.

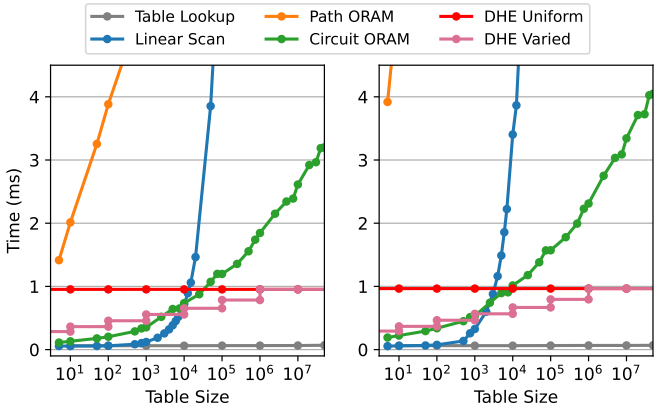
Comparison: Table I summarizes the complexity and the model accuracy of the different secure embedding generation methods. The computation complexity here is for a single embedding generation. The embedding table (storage-based) with protection applied does not scale well with the table size. The performance of DHE depends on k, assuming the DHE FC size is proportional to k. Note that $k \ll N$.

B. Performance Characterisation of Secure Embedding Generation Techniques

We implement, optimize, and deploy the three techniques described above in Intel SGX, considering security and performance, with implementation and optimization details in Sections V and VI-A. We measured the latency for processing one batch of embedding generation using each aforementioned technique.

TABLE I: Comparison of secure embedding generation methods. n is the table size; k is the no. of hash functions in DHE.

	Computation Complexity	Memory Space Complexity	Model Accuracy
Table: Linear Scan	$O(n)$	$O(n)$	no loss
Table: ORAM	$O(\log^2 n)$	$O(n)$	no loss
DHE	$O(k^2)$	$O(k^2)$	sized for no loss



(a) Embedding Dim. 16 (b) Embedding Dim. 64

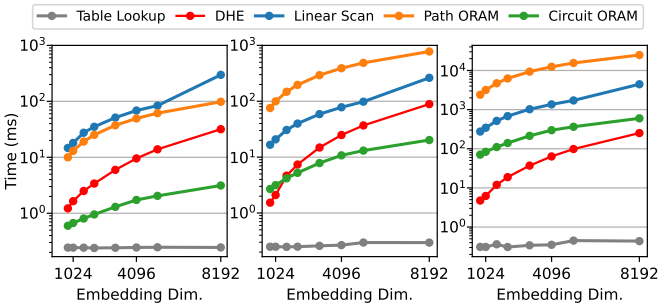
Fig. 4: Latency of the secure embedding generation techniques for different table sizes in DLRM (batch size = 32, 1 thread).

1) **DLRM:** We first study the embedding generation in DLRM. DLRM typically has a large number of embedding tables with different table sizes. Figure 4 shows the latency for different table sizes with fixed embedding dimensions of 16 and 64, in a single-threaded configuration with a batch size of 32. The evaluated DHE has k=1024 and a 3-layer FC.

Figure 4 shows that the optimal embedding generation technique in terms of latency is different for different table sizes for the two commonly used embedding dimensions. DHE has a constant cost independent of the table size, whereas the latency of linear scan and tree-based ORAM grows with the size of the table. Specifically, linear scan and Path ORAM become impractical for very large table sizes. Circuit ORAM is the fastest among the traditional oblivious computing schemes for large embedding table lookups, yet incurs a large latency.

We further optimize the DHE scheme by varying the DHE size (k and FC sizes) for different table sizes. Hence, we have two DHE schemes: *Uniform*, where the DHE model architecture is fixed for all tables, and *Varied*, where smaller tables can use smaller DHE models. The intuition behind Varied is that smaller table sizes need less powerful or expressive DHE models to generate embeddings and hence we do not necessarily need fix sized DHE for all tables. For the Varied DHE, we explored a few heuristic ways to change the DHE size, e.g., scaling down by 2 for every order of magnitude decrease in the table size, or scaling down linearly. We found the latter to match the baseline table model accuracy and thus adopted the linear scaling down approach for DHE sizes. It is possible to do a more exhaustive search for an even better parameter scaling down approach to optimize the DHE sizes for each table. Figure 4 shows that Varied DHE has a significantly smaller latency than the Uniform DHE depending on the table size.

Figure 4 shows that for small table sizes, linear scan is better than DHE and even tree-based ORAMs. This is because scanning a small range of indices is faster than executing the compute stacks in DHE or performing an ORAM access by reading/evicting a tree path. The figure suggests that a single



(a) Batch Size 1 (b) Batch Size 8 (c) Batch Size 256

Fig. 5: LLM embedding generation latency vs. embedding dimension, for various batch sizes, for a fixed table (vocabulary) size of 50257.

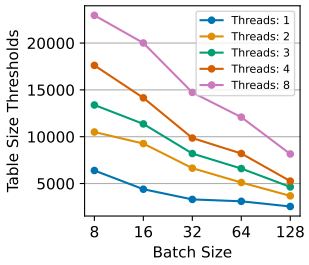


Fig. 6: Thresholds of table size obtained in offline profiling for embedding dim. 64.

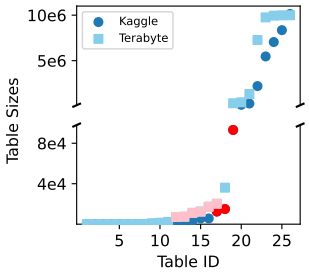


Fig. 7: The tables in the real DLRM datasets which fall within the range for hybrid are highlighted in red.

technique cannot provide optimal latency for all embedding table sizes, and there is an opportunity for a hybrid embedding generation approach for better performance. Morever, the techniques differ in metrics other than latency e.g., DHE has a much smaller memory footprint than a large embedding table, and so more DHE layers can fit into the memory of a given confidential computing system than raw tables, or tables protected by ORAM.

2) **LLM:** We also study these secure embedding generation techniques for Transformer-based LLMs. Figure 5 shows the trend of the token embedding latency for various embedding generation techniques, as the embedding dimension is changed, for a fixed vocabulary size of 50257 (as in GPT-2). We show different embedding generation batch sizes. An embedding generation batch size depends on the input query batch size and the LLM processing stage. The *prefill* stage will typically have larger batch sizes as it processes multiple token embeddings at once, whereas the *decoding* stage processes one token at a time. We use 16 threads for inference. The DHE is sized to match the baseline model *perplexity* (quality). For large batch sizes (e.g., prefilling input prompts), DHE performs best in terms of latency; for lower batch sizes (e.g., the decoding stage), the best technique is either DHE or Circuit ORAM depending on the exact batch size and also the embedding dimension i.e., the LLM model architecture. This points to a potential of using dual representation of the embedding table for best latency.

C. Hybrid Scheme for DLRM

DLRMs have many embedding tables of different sizes, and as shown in Figure 4, each table size is suited to different secure embedding generation techniques in terms of latency. We propose a hybrid protection scheme for DLRM in order to improve the model latency and throughput while ensuring security. For DLRM, either linear scan or DHE is the best-performing technique for any table size. We now describe how to allocate either linear scan or DHE to a given DLRM sparse feature for optimal performance, first considering a single model for latency and then later discussing co-located models for both latency and throughput. We use the term DHE to refer to both the Uniform and Varied variants.

1) **Optimizing Latency for a Single DLRM Inference:** We propose Algorithm 2 to minimize the latency of a single model on a machine, by selecting the best embedding generation technique for individual features. In our experiments, we consider DLRM model execution with sequential sparse feature processing.

Offline Profiling: The goal of profiling is to determine the performance of linear scan and DHE for different table (sparse feature) sizes on a given system, and find the *threshold* that determines which embedding generation technique to use: table sizes below this threshold use linear scan, and DHE otherwise. Here, we show the example profiling in our system (Section VI-A). We profile the latency for each technique across different table sizes with different execution configurations including batch sizes and CPU thread counts. For each profiled configuration, we plot the latency vs. table size curve for each of linear scan and DHE like in Figure 4(b), and take the point of intersection of the two curves to be the *threshold* between linear scan and DHE. Figure 6 shows the distribution of switching thresholds, for linear scan and DHE Uniform. The thresholds decrease as the batch size increases because DHE has better temporal locality and batch parallelism. On the other hand, as the number of threads increase, the linear scan improves its cache reuse of the table across several queries in multiple threads, so the thresholds increase. The profiling

Algorithm 2: Optimize DLRM Single-Model Latency

Offline:
1. Profile latencies with a given execution configuration to find (the range of) table (sparse feature) size thresholds for allocating either linear scan or DHE.
2. Convert the DHE representation of all below-threshold sparse features to a table, to be used by linear scan as required.

Online: Use the profiled thresholds at runtime to allocate the embedding generation technique (linear scan or DHE) based on the table (sparse feature) size and execution configuration, to achieve the best model latency.

Algorithm 3: Hybrid DLRM: Dynamic Scheduling

Func ScheduleDLRMTables (BatchSize, ThreadCount):
  CurrThreshold = Thresholds[BatchSize][ThreadCount];
  For each table T in the DLRM model:
    If TableSize(T) $\leq$ CurrThreshold: Assign T to LS
    If TableSize(T) $>$ CurrThreshold: Assign T to DHE

stage is of low effort and it can be completed in a few hours. It is done once per system for each embedding dimension.

The red points in Figure 7 show the DLRM tables in Kaggle and Terabyte datasets that can be allocated to either linear scan or DHE (Uniform). The tables/features below this (red-colored) range are always allocated to linear scan, and above always to DHE. For Kaggle and Terabyte, 7 and 9 out of a total of 26 tables (99.7% in terms of memory footprint of the table representation) will always benefit from the DHE format, respectively. The red features (3 tables for Kaggle and 6 tables for Terabyte) can choose to use linear scan or DHE dynamically depending on the configuration.

Offline Hybrid Model Training and Preparation: The profiling shows that the red features in Figure 7 can potentially either use linear scan of the embedding table or DHE at runtime, and hence inference needs access to a model that supports both. Naïvely, we can train separate end-to-end models for each execution configuration's threshold split. However, that would require extensive training time, and also increase the memory footprint. Instead, we propose to train a model with all sparse features in DHEs, and then *use the trained DHEs to create table representations* which store the DHEs' outputs for all valid inputs. If linear scan is faster than DHE for a given configuration, the table generated from trained DHE will be used. In this way, the model can be easily adapted to many system configurations without retraining.

Online Deployment: At inference time, the model including the hybrid table representations is loaded into memory. Based on the current execution configuration, we use the threshold from the profiling to choose the embedding generation technique for each feature. Then, each sparse feature is processed by either linear scan or DHE based on the table size. Algorithm 3 summarizes the online decision.

2) **Optimizing Throughput for Co-located DLRMs:** Typical data-centers may run many co-located models in parallel on the same system, to achieve high system throughput and high server utilization. The throughput of DLRM inference can be calculated as $1/\text{Latency} \times \text{Batch Size} \times \text{Number of Co-located Models}$. Meanwhile co-locating models can cause resource contention and interference among models, in terms of computation resources, caches, and memory bandwidth. Figure 8 shows the latency increase caused by co-locating multiple copies of a synthetic table that only uses one type of secure embedding generation technique. As a result, the threshold to determine the embedding generation technique may change in the case of co-location.

We study the effect of co-locating varying proportions of linear scan and DHE for embedding generation techniques

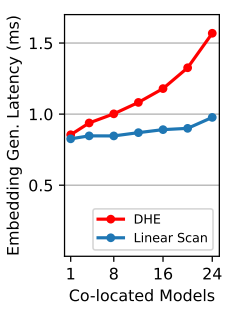


Fig. 8: Increasing co-location of embedding generation techniques.

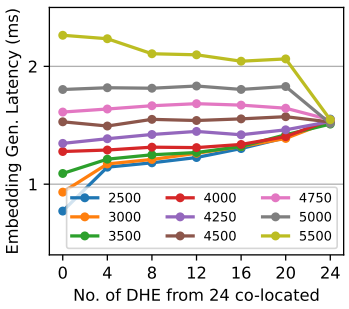


Fig. 9: Latency for different allocation of DHE and linear scan for fixed total co-location of 24 models. Legend indicates table sizes.

on total fixed $N = 24$ cores, across a variety of embedding table sizes. Figure 9 shows the embedding layer latency for embedding vector dimension 64, and one execution configuration (batch size 32, 1 thread). For smaller table sizes, allocating all tables as a linear scan gives the lowest latency (0 on the x-axis); but for larger table sizes (beyond 4500), DHE gives the best latency (24 on the x-axis). The co-located switching threshold of 4500 is close to the original single-model threshold of 3300 as shown in Figure 6 for batch size 32 and one thread. We generally observe that when all cores are assigned to the same embedding generation technique, the single-model threshold is still close to that for co-located models. Moreover, co-located models may be running differently sized sparse features and even some FC layers in parallel to ease the contention. Given the small potential benefit and the cost of optimization, we suggest using the single-model thresholds for deciding the embedding generation method even for co-located models.

3) **Training / Deployment:** So far, we discussed how to optimize the inference latency of secure DLRM by using a hybrid approach for embedding generation. To deploy such models, we need to first train DHE Uniform models to search DHE parameters that can match or exceed the baseline table accuracy on our dataset. Then we can optionally further optimize DHE sizes by training more models with DHE sizes that vary with table sizes. Thereafter, we can apply Algorithm 2 for inference. Because DHE has a deterministic access pattern in both forward and backward passes during training, the proposed all-DHE training should also be secure.

D. Hybrid Scheme for LLM

As shown in Figure 5, for a given LLM embedding dimension, either DHE or Circuit ORAM is the most favorable for latency of secure embedding generation, depending on the exact embedding generation batch size. Typically, only the decoding stage in LLMs will favor Circuit ORAM due to small batch sizes. We can choose to use either one dynamically by generating a table for ORAM from the outputs of a DHE-based embedding layer. Note that the memory overhead of ORAM

for a single embedding table may be high relative to the rest of the LLM model, especially for smaller language models.

In our LLM experiments, we fix the thread count as 16, and evaluate both DHE and Circuit ORAM for different batch sizes in the LLM prefill and decoding stages.

V. Security: Implementation and Analysis

A. Implementation of Embedding Generation Methods

Here, we describe our implementations of embedding generation methods, which avoid data-dependent memory access patterns and control flow that may lead to side-channel attacks. Our implementation uses Intel's Scalable SGX [53], first offered in the 3rd generation Ice Lake Xeon server processors. However, our choice of TEE does not specialize our design, and our approach can be adapted to other protected execution environments.

1) **Tree-based ORAM:** We adapted the open-source implementation in ZeroTrace [98], which has software-based Path ORAM and Circuit ORAM controllers, and uses data-oblivious functions (e.g. predicated execution using the conditional register–register `cmov` instruction) for implementing ORAM operations such as pos-map lookup to harden it against software side-channels such as cache-timing attacks. ZeroTrace was developed for the now obsolete Intel Client SGX edition, with a small 256 MB EPC memory for the enclave. Since Scalable SGX offers a large EPC memory (64 GB), we modified the ORAM controllers to include the entire memory inside the enclave using Gramine [40] to reduce context switches during execution and achieve better performance. As shown in Figure 10, including the whole ORAM tree in EPC (ZT–Gramine) reduces the average latency of the original ZeroTrace (ZT–Original) by 20% for Path ORAM and 60% for Circuit ORAM. We further optimize the implementation via enabling recursion by fixing bugs and inlining the `cmov` assembly function calls (ZT–Gramine–Opt), to achieve an additional latency reduction of 29% for Path ORAM and 54% for Circuit ORAM, on average for the table sizes shown in the figure. We make our best effort to optimize the ORAM implementation and use this optimized version for our evaluation.

For protecting an embedding table using ORAM, we match the ORAM block size to the embedding vector dimension. The bucket size of $Z = 4$ blocks, and stash sizes of 150 and 10 for Path ORAM and Circuit ORAM respectively are set according to previous work [98]. We enable recursion after 2^{12} blocks for Circuit ORAM, and 2^{16} blocks for Path ORAM, to empirically obtain the lowest latency. The pos-map tree reduction at each recursion level is $16\times$. Processing each item in the input batch is sequential since the internal ORAM structures must be updated sequentially and parallelism is not possible.

2) **Linear Scan of Table:** A linear scan of the entire table to retrieve an embedding hides the desired index. However, to avoid side-channels, we must make its implementation constant-time and also consider its data writing pattern to registers or memory. In the linear scan algorithm, there is an output tensor buffer to which the desired embedding from the table is copied obliviously as the table is scanned. We develop an efficient linear scan using AVX-512. We scan the entire embedding table for each input index in a batch, and obliviously set a flag to 1 once the desired row is reached. We expand the flag into an AVX bit-mask, and use an AVX *blend* instruction to obliviously copy the embedding from the table to the output buffer via AVX registers. When the embedding size is not a multiple of the AVX vector size, the left-over elements can use *masked* AVX loads/stores. This approach avoids using any data-dependent code branches.

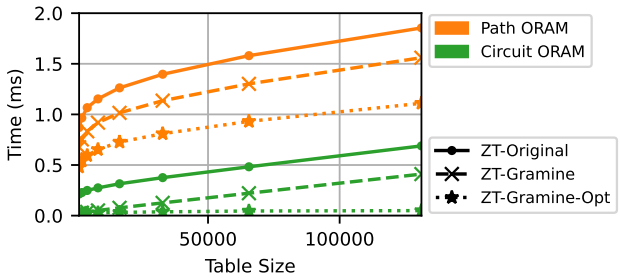


Fig. 10: Latency of a single lookup of Path/Circuit ORAM for embedding dimension 64.

3) **Deep Hash Embedding (DHE):** DHE consists of a simple hashing and scaling function, followed by fully-connected (FC) layers. The *hashing function* $((ax+b)\ mod\ m)\ mod\ p$ is implemented in a deterministic way: as an outer product, followed by element-wise addition and modulus operations. The transformation to real numbers is a simple scaling operation. The DHE *FC layers* consist of dense matrix multiplications and activation functions. For a given size, the dense matrix multiplications are by nature deterministic in terms of control/data flow. The *activation function* used is ReLU [3], which, in a scalar implementation, has a conditional branch that clamps the activation value to a minimum of zero. Deep learning libraries, such as PyTorch [83] that we use, typically harness vector/SIMD instructions to perform such operations e.g. `ReLU(x)=max(0,x)`. This has no branches as a hardware intrinsic, and hence is secure. We provide a proof-of-concept secure ReLU implementation that uses AVX-512.

B. Security Analysis

Table II summarizes how our embedding generation implementations are secured against different side-channels. Each column shows a signal that may depend on the secret. The

TABLE II: Security of embedding generation techniques.

	Secret-dependent Data Access	Secret-dependent Control Flow	
Potential Attacks	cache/memory side-channel; page fault controlled-channel	frontend/i-cache side-channel	
Table: non-secure	not protected	N/A*	
Table: ORAM	ORAM, linear scan	`cmov`	Protected
Table: Linear Scan	linear scan	branchless AVX	Protected
DHE (hash)	N/A	N/A	Protected
DHE (FC)	N/A	branchless AVX (ReLU)	Protected

* N/A means such a code pattern does not exist. Thus, it is **secure**.

second row shows how such signals can be *leaked* through practical side-channel attacks in TEEs. The content in Rows 4-7 show how our secure implementations *protect* the leakage source, where applicable.

Obfuscating Data Flow: To conceal input-dependent data access patterns existing in the table embedding representation, we use ORAM or linear scan. Note that the algorithmic security of ORAM (memory trace indistinguishablility) is well-established [101], [113], and linear scan is secure by definition as it touches all memory elements for each access. The software ORAM controller also uses linear scan for its internal structures including position map and stash, like in [98]. On the other hand, DHE performs the same computation for all inputs, and does not have input-dependent data access patterns.

Obfuscating Control Flow: To create constant-time algorithms without secret-dependent control flow, we use `cmov` and AVX instructions instead of conditional branches. Using `cmov` to replace branches with predicated execution is secure, as it operates at a register level and does not touch DRAM (see detailed analysis in §5.2 of [92]). Using AVX is secure for replacing code branches, since as a SIMD ISA, it performs optional computation and data movement on certain vector elements via mask predicates, and is inherently branchless. Orthogonal to these defenses, we remark that mathematical operations such as matrix multiplication and element-wise transformations (e.g. multiplicative scaling) have control flows independent of inputs by nature, and for a given input shape, follow deterministic code-paths in ML frameworks such as PyTorch [83].

Security of the Hybrid Scheme: The DLRM hybrid scheme consists of linear scan and DHE, both of which have been shown to be individually secure in theory and implementation above. As long as the decision to choose the technique that gives the lowest latency for a given sparse feature is independent of user inputs, the hybrid scheme should have the same security guarantees as the constituent techniques. The system-level decision to choose the embedding generation technique for each sparse feature in a DLRM model only depends on the system execution configuration (batch size and thread count in our setup), as shown in Algorithm 3. The decision is unaffected by user input/output/intermediate values of the model; hence, the hybrid scheme does not leak information about the input. Similarly, the decision to choose DHE or Circuit ORAM in LLM generation depends on only the embedding generation batch size, which in turn depends on the inference query batch size, LLM stage (prefill or decoding) and token counts, none of which we hide in our threat model. Note that unlike DLRM, we fix the thread count in our LLM experiments.

C. End-to-End Protection of the Whole ML Model

This work focuses on protecting embedding generation from side-channel attacks. However, for enabling end-to-end protection, we also need to secure the data-dependent memory access patterns, if any, in other layers.

DLRMs, as shown in Figure 1, have standalone FC layers (bottom & top), which have a deterministic memory access pattern except for the activation layers (i.e., ReLU, which we secured). The final click probability is obtained via *sigmoid* which is a mathematical deterministic operation. The interaction layer is deterministic as it is either the concatenation *or* an all-to-all inner product of the dense and sparse embeddings.

In LLMs, beyond the token embedding generation, there are layers of FC, attention, and normalization layers, all of which have deterministic data and control flow. Some of these other layers have interspersed activation functions like SoftMax and GeLU [48], which involve deterministic mathematical operations and are oblivious. During greedy sampling in generation, the search over the output logits to determine the most probable token (`argmax`) can be written as a linear scan that copies the maximum value obliviously using `cmov`. While there are functions such as causal attention masking that depend on the prompt length, we do not aim to hide the length.

The overhead of securing layers other than embedding generation is small, e.g., the overhead of securing `argmax` in LLMs is less than 0.4% of the total generation latency. On the other hand, securing embedding generation is clearly the biggest challenge for end-to-end protection, as naïve protection with traditional methods can lead to performance and memory overheads, especially for DLRMs. As we will show in the evaluation, our secure embedding generation method based on DHE enables practical and efficient end-to-end side-channel protection for both DLRM and LLM.

VI. Evaluation

A. Methodology

1) System Setup: We run our experiments on a server-class processor, detailed in Table III. We are limited to 64 GB memory in our TEE setup, but Scalable SGX can support up to 1 TB secure memory via dual sockets.

TABLE III: System Configuration.

Processor	Intel Xeon Gold 6348, Ice Lake, 3.50 GHz, 28 Cores 56 Threads
SIMD	up to AVX-512
Cache	42 MB LLC, non-inclusive
DRAM	DDR4-3200 128 GB, 8 channels
TEE	Intel Scalable SGX, EPC Size 64 GB

We use PyTorch 2.0 [83], for inference and training. We adapt existing implementations for DLRM [80], LLMs i.e., HuggingFace [120], and DHE [50]. We implement linear scan and tree-based ORAM, based on ZeroTrace [98], as C++ extensions to PyTorch using pybind [58]. Thus, for all techniques, we are able to run an end-to-end inference using the PyTorch framework. To port our inference flow to the TEE, we use Gramine [40], a lightweight library OS that enables the execution of unmodified Linux binaries inside SGX.

2) DLRM: Model Architecture and Datasets: Table IV shows the details of the Criteo datasets and the corresponding models we use, inspired from prior work [50]. For DHE Varied, the DHE sizes are scaled down .125× for every order of magnitude decrease in the table size, starting from 1e7.

TABLE IV: DLRM datasets and model information.

Dataset	Dense Feat.	Sparse Feat.	Emb. Size	FC Bottom & Top Sizes	DHE (Uniform) Parameters
Criteo Kaggle [59]	13	26	16	512-256-64-16 512-256-1	$k = 1024$ FC: 512-256-16
Criteo Terabyte [21]	13	26	64	512-256-64 512-512-256-1	$k = 1024$ FC: 512-256-64

Beyond these Criteo datasets, we also analyze the performance of a DLRM model whose sizes are based on an open-source DLRM dataset [78] released by Meta, which contains synthetic embedding lookup traces that mimic real production data. We use the 2022 traces to determine the sizes of 788 sparse embedding feature tables, which go up to 4e7 entries (unlike Criteo which only go up to 1e7).

Metrics: We evaluate and compare the proposed hybrid scheme for DLRM to homogeneously secure models (based on a single secure technique, DHE or traditional memory protection), via the metrics of model accuracy, model size (i.e. memory footprint), and performance (i.e. model latency). We also analyze the system-level throughput of co-located models.

3) LLM: Model Architecture and Datasets: We use GPT-2 medium [90], a 355M-parameter model with an internal embedding dimension of 1024, and a vocabulary (embedding table) size of 50257. Instead of training a DHE-based model from scratch, we finetune the pretrained model over the OpenWebText dataset [37]. The DHE we use has 4 FC layers, and both the internal FC sizes and k are twice the embedding dimension i.e. $2\times1024{=}2048$ for GPT-2 medium.

Metrics: We study standard LLM performance metrics i.e. the prefill latency or the time to first token (TTFT), the decode latency or the time between tokens (TBT), the decoding throughput (tokens/s), and the end-to-end generation latency. For comparing the model quality, we use perplexity, which measures the confidence a model has in its predictions. We also study the overhead in memory footprint i.e. model size, due to introducing secure embedding generation in LLMs.

B. DLRM Results on Criteo Datasets

Throughout this section, unless otherwise stated, the performance results are based on an execution configuration of 1 thread per model and a batch size of 32.

1) Accuracy: We first trained DLRMs using the table embedding representation to obtain the baseline accuracy (i.e. that of the linear scan or tree-based ORAM). We then trained the whole DLRM with DHE layers with the goal of matching the baseline accuracy. The attainable accuracy depends on the number of hashes in DHE i.e. the k parameter, and the DHE FC sizes, but both also impact the latency. Prior work from industry [2], [44] states that even small accuracy degradations are undesirable due to their real-world impact. Table V shows the accuracy achieved for the baseline table and both DHE versions described earlier: Uniform and Varied. **With proper DHE hyperparameters, the accuracy of the DLRM with DHE matches that of embedding tables.**

2) Memory Footprint: We approximate the memory footprint of the non-secure baseline, and the secure embedding generation techniques, by considering the model size; see Table VI. **The DHE and hybrid models are orders of magnitude smaller than the table/ORAM-based baseline models.** Given the small size of the hybrid models, the TEE secure memory can hold thousands of such models; whereas TEE memory capacity will fall short for multiple table-based models. The tables represented as Tree-ORAM are even larger (more than $3\times$) due to the tree having dummy blocks, and the pos-map auxiliary structure having its own trees due to recursion. The Path and Circuit ORAM models have negligible difference in memory. Relative to Tree-ORAM, the DHE/Hybrid variants occupy between $101-278\times$ less memory for Kaggle, and $554-1116\times$ less memory for Terabyte.

TABLE V: DLRM Model Accuracies.

	Table	DHE Uniform	DHE Varied
Kaggle	78.82%	78.82%	78.82%
Terabyte	80.96%	80.97%	80.96%

TABLE VI: DLRM model memory footprint for various techniques. Memory size relative to table (◎) representation is shown.

	Model Memory Footprint (MB)	
	Kaggle	Terabyte
Table	2062.7 (◎)	11999.2 (◎)
Tree-ORAM	6903.6 (327.4%)	40421.4 (336.9%)
DHE Uniform	68.2 (3.31%)	73.0 (0.61%)
DHE Varied	33.4 (1.62%)	40.5 (0.34%)
Hybrid Uniform	26.9 (1.30%)	45.6 (0.38%)
Hybrid Varied	**24.9** (1.20%)	**36.2** (0.30%)

3) Single Model Latency: In Section IV-C1, we described our scheme based on profiling and obtaining thresholds using small benchmarks. We first aim to verify the effectiveness of the profiled thresholds for real datasets. At each of our selected execution configurations, we sweep across many threshold values and observe the end-to-end latency of the model. We compare the allocation for the empirically achieved lowest latency to that suggested by the profiled database. Analyzing the Hybrid Varied model, we observe that profiled database is near-optimal (threshold is incorrect by maximum ±1 table) for 88% of our execution configurations for Kaggle, and 84% for Terabyte, despite execution noise. This shows that the **profiled database for a single table gives near-optimal latency with real datasets having multiple tables, over many execution configurations**. Figure 11 shows the sweep over thresholds in one particular execution configuration of the Hybrid Varied model. In this case, the profiling-suggested and best-empirical table allocation were an exact match.

We show the end-to-end inference latency under the different protection techniques, in Table VII. The index-lookup baseline has a small latency but is not secure. Securing the DLRM via purely linear scan increases the latency from milliseconds to the seconds range. From the tree-based ORAMs, Circuit ORAM performs much better than Path ORAM, as also shown in Section IV-B1. As such, Circuit ORAM is our most competitive baseline. We now look at the latencies for DHE

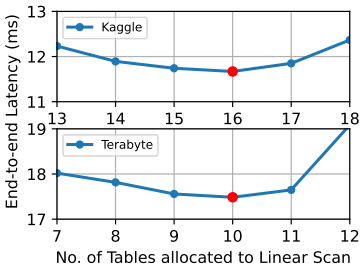


Fig. 11: DLRM single model latency for different allocation of sparse features, best latency highlighted (Hybrid Varied).

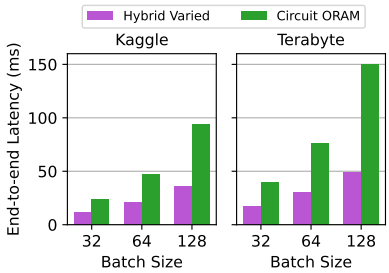


Fig. 12: End-to-end latency of DLRM models as the batch size is increased.

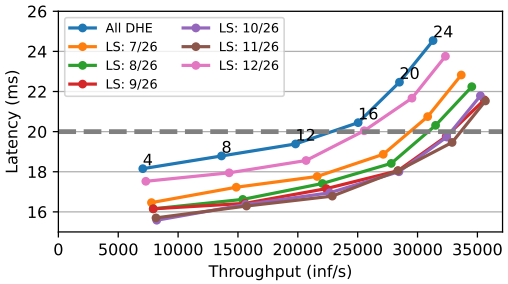


Fig. 13: Performance of DLRM during increasing model co-location with different sparse feature allocation (DHE Varied and Hybrid Varied shown for Terabyte). The target latency is 20 ms.

TABLE VII: DLRM end-to-end model latencies. Speed-ups or slow-downs relative to Circuit ORAM (⋄) are indicated (↑, ↓).

	End-to-end Latency (ms)	
	Kaggle	**Terabyte**
Index Lookup (non-secure)	1.58	2.02
Linear Scan	7970.6	44996.4
Path ORAM	323.1	761.9
Circuit ORAM	23.9 (⋄)	39.9 (⋄)
DHE Uniform	29.4 (0.81× ↓)	30.7 (1.30× ↑)
DHE Varied	17.4 (1.37× ↑)	20.0 (1.99× ↑)
Hybrid Uniform	13.2 (1.82× ↑)	21.1 (1.89× ↑)
Hybrid Varied	**11.9** (2.01× ↑)	**17.5** (2.28× ↑)

and Hybrid models, which have the same accuracies as the table/ORAM representation. Relative to Circuit ORAM, DHE Uniform offers a speed-up only for the Terabyte model, but is overkill for the smaller table sizes. DHE Varied reduces the DHE layers sizes, and hence is able to speed-up the performance relative to Circuit ORAM by 1.37× for Kaggle and 1.99× for Terabyte. The Hybrid models introduce linear scan as a more efficient technique for the smallest tables in the models, and hence further reduce the pure-DHE latencies by an average of 1.85× for Kaggle, and 1.30× for Terabyte. On average, the Hybrid Uniform model achieves a 1.86× improvement and Hybrid Varied achieves a 2.14× improvement over the Circuit ORAM scheme.

Figure 12 shows how the end-to-end latency varies as the batch size is increased. The figure shows that the **Hybrid scheme scales better than ORAM with higher batch sizes**, since ORAM has to issue sequential accesses for each query in the batch, whereas DHE can take advantage of weight reuse. At batch size 128, Hybrid Varied is better than Circuit ORAM by 2.61× for Kaggle and 3.08× for Terabyte, better than the improvement for batch size 32 in Table VII.

Even though the best performing secure scheme (Hybrid Varied) is 7.5× and 8.8× slower than the non-secure scheme for Kaggle and Terabyte respectively, the DHE-based protection still satisfies the typical Service Level Agreement (SLA) target latencies of industry models, which can go up to 100s of ms (see Table II in [43]).

4) Co-located Model Latency-Throughput: We co-locate multiple hybrid models on our system to maximize the system throughput and study the latency-throughput characteristics. Figure 13 shows the increasing co-location of DHE and Hybrid Varied models (only Terabyte shown for brevity). The results show that the linear scan/DHE **allocation based on the single-model thresholds can still effectively achieve the best latency-throughput curve** (10/26 linear scan tables for Terabyte; 16/26 for Kaggle, not shown). Moreover, the proposed hybrid protection can significantly improve the system throughput compared to only using DHE for all tables. Assuming a SLA target latency of 20ms, **Hybrid Varied significantly increases the latency-bounded throughput compared to DHE Varied**, from 33.2K inferences/s to 52.2K for Kaggle (1.6×), and from 22.8K to 32.8K for Terabyte (1.4×). Note that co-locating large number of ORAM-based models is not possible in TEE memory due to their huge memory footprint.

C. DLRM Results on Meta Dataset

We analyze the performance of the embedding generation layers of a DLRM model based on the 2022 Meta dataset [78]. This has many more tables (788) that are also larger, hence more representative of a production model. We do not train the model since there is no ground truth available; we assume that the DHE size of Criteo Terabyte will be sufficient to result in an accuracy that matches that of a baseline table

TABLE VIII: Results on a DLRM from the Meta Dataset: embedding generation latency and memory footprint. Speed-ups compared to Circuit ORAM (⋄) are indicated (↑). Memory size relative to table (◎) representation is shown.

	Latency (ms)	**Memory (MB)**
Index Lookup (non-secure)	52.1	931335.7 (◎)
Linear Scan	3484553.6	same as above
Path ORAM	28413.6	3090046.3 (331.8%)
Circuit ORAM	1347.0 (⋄)	same as above
DHE Uniform	906.9 (1.49× ↑)	2050.9 (0.22%)
DHE Varied	647.9 (2.08× ↑)	1324.5 (0.14%)
Hybrid Uniform	581.4 (2.32× ↑)	**1199.0** (0.13%)
Hybrid Varied	**560.2** (2.40× ↑)	1219.5 (0.13%)

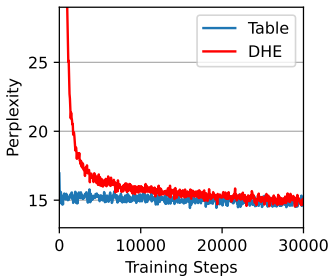


Fig. 14: LLM perplexity during finetuning for GPT-2 medium.

Fig. 15: Latency (ms) of the GPT-2 medium LLM. Speed-up (↑) or slow-down (↓) relative to Circuit ORAM (⋄) is indicated. *Batch* refers to the inference request size and will scale by 256× to give the embedding generation batch size in prefill.

	Prefill/TTFT Batch=1	**Decode/TBT Batch=1**	**Prefill/TTFT Batch=8**	**Decode/TBT Batch=8**	**Prefill/TTFT Batch=12**	**Decode/TBT Batch=12**
Index Lookup (non-secure)	183.7	37.2	1942.6	74.2	3114.4	117.6
Linear Scan	534.7	57.1	5305.5	95.8	7982.0	139.6
Path ORAM	3355.5	51.5	27350.8	174.6	41204.9	268.7
Circuit ORAM	250.8 (⋄)	**38.6** (⋄)	2499.5 (⋄)	78.6 (⋄)	3980.2 (⋄)	126.9 (⋄)
DHE	**190.0** (1.32×↑)	38.9 (0.99×↓)	**2023.9** (1.24×↑)	**76.1** (1.03×↑)	**3242.5** (1.23×↑)	**118.9** (1.07×↑)

representation. We set an embedding dimension of 64 as in the Criteo Terabyte model.

Table VIII shows the embedding generation latency in this DLRM model for various techniques, for a batch size of 32 and 1 thread. We calculate the overall latency by executing few tables at a time in our limited SGX memory. The most competitive secure baseline is again Circuit ORAM which executes the layers in 1.35s. The best DHE-based scheme i.e. Hybrid Varied, improves this latency by 2.40×. Table VIII also shows that ORAM exacerbates the already large memory footprint of the table representation (910 GB) by 3.32×, and is impractical to deploy. The DHE based models are extremely memory-efficient in comparison: Hybrid Varied is only 1.2 GB, a reduction of over 2500×.

D. LLM Results

1) Accuracy: We finetuned the pretrained GPT-2 medium model over the OpenWebText dataset, first with the original table to obtain the baseline perplexity, and then with the table replaced by DHE. Figure 14 shows the test set perplexity (which is proportional to the test loss) after finetuning for a few thousand iterations. **The difference in the lowest perplexity achieved is small** i.e., the perplexity is 14.6 for the table compared to 15.0 for DHE (only a 2.7% deterioration in perplexity). To achieve good LLM utility with DHE, we found it is critical to finetune the entire model, not just the embedding layer.

2) Latency: We evaluate the LLM performance on an input prompt of 256 tokens (the prefill size), and an output generation (decoding) length of 128 tokens. We evaluate three inference batch sizes: 1, 8 and 12 (the largest we could execute in our SGX memory). Table 15 shows the performance of GPT-2 medium with various embedding generation techniques. The table shows that the finetuned DHE-based model generally gives the best latency for both prefill and decode (albeit at a very slight drop in perplexity). The only exception occurs during the decoding stage for very small (~ 1) batch sizes, whereby Circuit ORAM, again the best baseline scheme, has a slight edge. For best performance in this scenario, a hybrid embedding scheme based on both DHE (for prefill) and ORAM (for decoding) can be used. One the other hand, for large input batch sizes, **DHE outperforms Circuit ORAM by up to 1.07× in decoding**, due to its superior batch parallelism.

For prefill, DHE is always better than ORAM due to typically large token (batch) counts; **on average, DHE improves the prefill time by 1.26×relative to Circuit ORAM**. In terms of the end-to-end generation latency, the performance of DHE is similar to the non-secure model, with an overhead of 2–5% depending on the batch size. In contrast, the multiplicative overhead of decoding in Circuit ORAM at large batch sizes can lead to large end-to-end overheads, up to 12%.

3) Memory Footprint: Compared with the original embedding table of 196.3 MB, the introduction of DHE adds 56.0 MB of parameters to the GPT-2 medium model (1353.5 MB), i.e., only a 4% memory overhead. On the other hand, the ORAM representation of the embedding table (513.6 MB) does add a 38% overhead. Note that these memory space overheads will be smaller for larger LLM models.

VII. Related Work

Protection for Embedding Tables: Embedding generation methods that are alternative to table lookup have been recently proposed as a way to improve efficiency [61], [125]. We propose to re-purpose DHE [61] for secure embedding generation as it has a deterministic memory access pattern. DHE was used by MP-Rec [50], along with table representation as part of a multi-path recommendation system that optimizes a heterogeneous platform based on accuracy, memory, and performance; however, their work did not consider security. Tensor Train (TT) decomposition [125] was used to decompose the large embedding tables into multiple smaller matrices to use a mix of lookup and computation; however, accessing the resulting matrices still leaks information via indices, and it is not secure. Look-Ahead ORAM [91] was proposed for DLRM training by optimizing ORAM based on known future training samples, whereas we focus on inference in our study where inputs are not known and thus it is more challenging.

Alternative PPML techniques such as MPC and HE build upon the one-hot matrix multiplication based embedding table lookup [64], [116], but instead of these crypto-based techniques, we use the much better performing TEEs for PPML.

Software ORAM in TEEs: In this study, we use two types of ORAMs, Path ORAM [101] and Circuit ORAM [113]. There exist other ORAM proposals as well with different performance characteristics [13], [49], [107]. However, we only use ORAM to establish a reasonable software baseline,

and the proposed DHE approach for security outperforms ORAM for all embedding table sizes that we found in today's DLRMs. Even for LLMs, DHE outperforms ORAM for large and practical batch sizes.

Mitigation of Side-Channel Attacks: Defenses have been proposed for each of the architecture components to mitigate side-channel attacks. In *caches and TLBs*, static/dynamic partitioning [20], [23], [26], [72], [97], [114], [117], [130] and randomized set mapping [23], [88], [89], [119] have been introduced. In the *processor front-end*, side-channel attacks have been demonstrated to leak the secret-dependent control flow [22], [33], [71], [95], [124]; partitioning [110], [127] and randomization [69] has been proposed to mitigate the side channels. For the attack surfaces in *the memory controller, memory buses and DRAMs*, mechanisms have been proposed to shape the victim's memory access pattern to mitigate contention-based attacks [24], [99], [135], [136] and ORAM in the memory controller [76], [94], [111] have been proposed to defend against an attacker who can observe the address of memory accesses. Yet, hardware-based ORAMs also have a high latency overhead, 10–50× per access [34], [76], [94]. For the side-channel relying on page fault in SGX, isolation and/or obfuscation schemes have also been proposed [4], [67], [100]. However, these mitigation mechanisms focus only on a subset of the attack surfaces and require hardware changes not available today. Efficient solutions that cover all side-channel attacks for ML models are needed.

Another possible mitigation is to randomize the layout of the address space. However, static randomization with a random offset does not provide enough entropy [1], [32], [75]. Morpheus [36], [46] supports frequent re-randomization, but side-channel is out of its scope. OBFUSCURO [4] and Raccoon [92] use ORAM to randomize the data flow and incur 40-400× performance overhead depending on the workload. MoLE [67] relocates data with random numbers, incurring 10× performance overhead while still leaving a small attack window.

VIII. Conclusion

In this paper, we study secure embedding generation methods for hiding information leakage through embedding table accesses in DLRMs and LLMs. We propose to use a computation-based method (DHE) to secure memory access pattern of embedding tables, and use it in conjunction with traditional methods like linear scan for high performance. Even though DHE is computation-intensive and is not as fast as non-secure embedding table lookups and thus is not widely deployed today, when memory access patterns need to be protected, DHE shows its advantage. We implement secure prototypes and demonstrate competitive performance for both DLRM and LLMs with a negligible or no accuracy drop.

Acknowledgment

The authors would like to thank Samuel Hsia and Hanieh Hashemi for their helpful technical discussions. We thank the reviewers for their constructive comments that significantly enhanced the manuscript.

This project is partially supported by Commonwealth Cybersecurity Initiative, by 4-VA, a collaborative partnership for advancing the Commonwealth of Virginia, by the National Science Foundation (NSF) under grants CCF-2153748 and CCF-2118709, and by the Air Force Office of Scientific Research under award number FA9550-22-1-0548. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the United States Air Force.

References

[1] "Address space layout randomization," https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[2] B. Acun, M. Murphy, X. Wang, J. Nie, C.-J. Wu, and K. Hazelwood, "Understanding training efficiency of deep learning recommendation models at scale," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, Feb. 2021. [Online]. Available: https://doi.org/10.1109/hpca51647.2021.00072

[3] A. F. Agarap, "Deep learning using rectified linear units (relu)," *arXiv preprint arXiv:1803.08375*, 2018.

[4] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine on intel sgx," in *Network and Distributed System Security Symposium*, 2019.

[5] M. Alam and D. Mukhopadhyay, "How secure are deep learning algorithms from side-channel based reverse engineering?" in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–2.

[6] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," ARM, White Paper, 2004.

[7] AMD, "AMD Memory Encryption," https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf, 2021.

[8] R. Anil, S. Gadanho, D. Huang, N. Jacob, Z. Li, D. Lin, T. Phillips, C. Pop, K. Regan, G. I. Shamir *et al.*, "On the factory floor: Ml engineering for industrial-scale ads recommendation models," *arXiv preprint arXiv:2209.05310*, 2022.

[9] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 42–56. [Online]. Available: https://doi.org/10.1145/3352460.3358310

[10] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[11] J. Carter and M. N. Wegman, "Universal classes of hash functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, p. 143–154, Apr. 1979. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(79)90044-8

[12] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[13] Z. Chang, D. Xie, and F. Li, "Oblivious ram: A dissection and experimental evaluation," *Proc. VLDB Endow.*, vol. 9, no. 12, p. 1113–1124, 2016. [Online]. Available: https://doi.org/10.14778/2994509.2994528

[14] Y. Che and R. Wang, "Dnncloak: Secure dnn models against memory side-channel based reverse engineering attacks," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, 2022, pp. 89–96.

[15] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir *et al.*, "Wide & deep learning for recommender systems," in *Proceedings of the 1st workshop on deep learning for recommender systems*, 2016, pp. 7–10.

[16] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "Secureme: A hardware-software approach to full system security," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 108–119. [Online]. Available: https://doi.org/10.1145/1995896.1995914

[17] C. C. Consortium, "Confidential Computing Consortium," https://confidentialcomputing.io/about/.

[18] V. Costan and S. Devadas, "Intel sgx explained," Cryptology ePrint Archive, Paper 2016/086, 2016, https://eprint.iacr.org/2016/086. [Online]. Available: https://eprint.iacr.org/2016/086

[19] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[20] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 857–874.

[21] Criteo AI Lab, "Criteo 1tb click logs dataset." [Online]. Available: https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/

[22] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: Security vulnerabilities in processor frontends," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 53–66.

[23] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 346–359.

[24] P. W. Deutsch, Y. Yang, T. Bourgeat, J. Drean, J. S. Emer, and M. Yan, "Dagguise: mitigating memory timing side channels," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 329–343.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "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, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[26] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–21, 2012.

[27] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 972–984.

[28] S. B. Dutta, H. Naghibijouybari, A. Gupta, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Spy in the gpu-box: Covert and side channel attacks on multi-gpu systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[29] M. J. Dworkin, "Recommendation for block cipher modes of operation :," Tech. Rep., 2010. [Online]. Available: https://doi.org/10.6028/nist.sp.800-38e

[30] G. G. et al., "llama.cpp," https://github.com/ggerganov/llama.cpp, 2024.

[31] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 190–202.

[32] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[33] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

[34] C. W. Fletcher, L. Ren, A. Kwon, M. v. Dijk, E. Stefanov, D. Serpanos, and S. Devadas, "A low-latency, low-area hardware oblivious ram controller," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2015, pp. 215–222.

[35] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, ser. STC '12. New York, NY, USA: ACM, 2012, pp. 3–8. [Online]. Available: http://doi.acm.org/10.1145/2382536.2382540

[36] M. Gallagher, L. Biernacki, S. Chen, Z. B. Aweke, S. F. Yitbarek, M. T. Aga, A. Harris, Z. Xu, B. Kasikci, V. Bertacco *et al.*, "Morpheus: A vulnerability-tolerant secure architecture based on ensembles of moving target defenses with churn," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 469–484.

[37] A. Gokaslan and V. Cohen, "Openwebtext corpus," http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[38] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," *J. ACM*, vol. 43, no. 3, p. 431–473, may 1996. [Online]. Available: https://doi.org/10.1145/233551.233553

[39] Google, "Vertex AI SDK for Python: List and count tokens," https://cloud.google.com/vertex-ai/generative-ai/docs/multimodal/list-token, 2024.

[40] Gramine Project, "Gramine – a Library OS for Unmodified Applications," https://gramineproject.io/, 2023.

[41] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security Privacy*, vol. 14, no. 6, pp. 54–62, Nov 2016.

[42] S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. D. Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, H. S. Behl, X. Wang, S. Bubeck, R. Eldan, A. T. Kalai, Y. T. Lee, and Y. Li, "Textbooks are all you need," 2023. [Online]. Available: https://arxiv.org/abs/2306.11644

[43] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 982–995.

[44] U. Gupta, S. Hsia, J. Zhang, M. Wilkening, J. Pombra, H.-H. S. Lee, G.-Y. Wei, C.-J. Wu, and D. Brooks, "RecPipe: Co-designing models and hardware to jointly optimize recommendation quality and performance," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3466752.3480127

[45] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 488–501.

[46] A. Harris, T. Verma, S. Wei, L. Biernacki, A. Kisil, M. T. Aga, V. Bertacco, B. Kasikci, M. Tiwari, and T. Austin, "Morpheus ii: A risc-v security extension for protecting vulnerable software and hardware," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2021, pp. 226–238.

[47] H. Hashemi, W. Xiong, L. Ke, K. Maeng, M. Annavaram, G. E. Suh, and H.-H. S. Lee, " Private Data Leakage via Exploiting Access Patterns of Sparse Features in Deep Learning-based Recommendation Systems," in *Workshop: Trustworthy and Socially Responsible Machine Learning at Neural Information Processing Systems*, 2022.

[48] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," 2023. [Online]. Available: https://arxiv.org/abs/1606.08415

[49] T. Hoang, M. O. Ozmen, Y. Jang, and A. A. Yavuz, "Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 1, 2019.

[50] S. Hsia, U. Gupta, B. Acun, N. Ardalani, P. Zhong, G.-Y. Wei, D. Brooks, and C.-J. Wu, "MP-Rec: Hardware-Software Co-Design to Enable Multi-Path Recommendation," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 449–465. [Online]. Available: https://doi.org/10.1145/3582016.3582068

[51] W. Hua, Z. Zhang, and G. E. Suh, "Reverse engineering convolutional neural networks through side-channel information leaks," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[52] Intel, "White Paper — Intel Total Memory Encryption," https://www.intel.com/content/dam/www/central-libraries/us/en/documents/white-paper-intel-tme.pdf.

[53] Intel, "Supporting Intel SGX on Multi-Socket Platforms," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/supporting-intel-sgx-on-mulit-socket-platforms.pdf, 2021.

[54] Intel, "Intel Trust Domain Extensions White Paper," https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf, 2023.

[55] Intel, "IPEX-LLM," https://github.com/intel-analytics/ipex-llm, 2024.

[56] Intel Corporation, "Intel SGX Developer Guide," 2023. [Online]. Available: https://download.01.org/intel-sgx/latest/linux-latest/docs/Intel_SGX_Developer_Guide.pdf

[57] R. Jain, S. Cheng, V. Kalagi, V. Sanghavi, S. Kaul, M. Arunachalam, K. Maeng, A. Jog, A. Sivasubramaniam, M. T. Kandemir *et al.*, "Optimizing cpu performance for recommendation systems at-scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[58] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2017, https://github.com/pybind/pybind11.

[59] O. C. Jean-Baptiste Tien, joycenv, "Display advertising challenge," 2014. [Online]. Available: https://kaggle.com/competitions/criteo-display-ad-challenge

[60] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[61] W.-C. Kang, D. Z. Cheng, T. Yao, X. Yi, T. Chen, L. Hong, and E. H. Chi, "Learning to Embed Categorical Features without Embedding Tables for Recommendation," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, ser. KDD '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 840–850. [Online]. Available: https://doi.org/10.1145/3447548.3467304

[62] W.-C. Kang, D. Z. Cheng, T. Yao, X. Yi, T. Chen, L. Hong, and E. H. Chi, "Learning to embed categorical features without embedding tables for recommendation," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, 2021, pp. 840–850.

[63] L. Ke, U. Gupta, M. Hempstead, C.-J. Wu, H.-H. S. Lee, and X. Zhang, "Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 141–154.

[64] J.-Y. Kim, S. Park, J. Lee, and J. H. Cheon, "Privacy-preserving embedding via look-up table evaluation with fully homomorphic encryption," in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 24 437–24 457. [Online]. Available: https://proceedings.mlr.press/v235/kim24ab.html

[65] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "Crypten: Secure multi-party computation meets machine learning," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[66] D. Kohlbrenner and H. Shacham, "On the effectiveness of mitigations against floating-point timing channels," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 69–81.

[67] F. Lang, W. Wang, L. Meng, J. Lin, Q. Wang, and L. Lu, "Mole: Mitigation of side-channel attacks against sgx via dynamic data location escape," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 978–988.

[68] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys'20, 2020.

[69] J. Lee, Y. Ishii, and D. Sunwoo, "Securing branch predictors with two-level encryption," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–25, 2020.

[70] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Zhenghong Wang, "Architecture for protecting critical secrets in microprocessors," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 2–13.

[71] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector:{High-Precision} branch target injection attacks exploiting the indirect branch predictor," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2137–2154.

[72] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2016, pp. 406–418.

[73] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[74] Y. Liu, D. Dachman-Soled, and A. Srivastava, "Mitigating reverse engineering attacks on deep neural networks," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2019, pp. 657–662.

[75] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 280–291.

[76] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "Phantom: Practical oblivious computation in a secure processor," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 311–324.

[77] 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 *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. NY, USA: ACM, 2013.

[78] Meta, "Embedding lookup synthetic dataset," https://github.com/facebookresearch/dlrm_datasets, 2022.

[79] H. Naghibijouybari, E. M. Koruyeh, and N. Abu-Ghazaleh, "Microarchitectural attacks in heterogeneous systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–40, 2022.

[80] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[81] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious {Multi-Party} machine learning on trusted processors," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 619–636.

[82] OpenAI, "How to count tokens with Tiktoken," https://cookbook.openai.com/examples/how_to_count_tokens_with_tiktoken, 2024.

[83] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[84] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for Cross-CPU attacks," in *25th USENIX security symposium (USENIX security 16)*, 2016, pp. 565–581.

[85] O. Press and L. Wolf, "Using the output embedding to improve language models," *EACL 2017*, p. 157, 2017.

[86] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+scope: Overcoming the observer effect for high-precision cache contention attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2906–2920. [Online]. Available: https://doi.org/10.1145/3460120.3484816

[87] PyTorch, "torch.nn.Embedding," https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html.

[88] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[89] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 360–371.

[90] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[91] R. Rajat, Y. Wang, and M. Annavaram, "LAORAM: A Look Ahead ORAM Architecture for Training Large Embedding Tables," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589111

[92] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital Side-Channels through obfuscated execution," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 431–446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[93] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 26–39.

[94] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 571–582.

[95] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 361–374.

[96] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "Xonn:xnor-based oblivious deep neural network inference," in *USENIX Security 19*, 2019, pp. 1501–1518.

[97] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 2021, pp. 37–49.

[98] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace : Oblivious memory primitives from intel SGX," in *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018. [Online]. Available: https://doi.org/10.14722/ndss.2018.23239

[99] A. Shafiee, A. Gundu, M. Shevgoor, R. Balasubramonian, and M. Tiwari, "Avoiding information leakage in the memory controller with fixed service policies," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 89–101.

[100] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-sgx: Eradicating controlled-channel attacks against enclave programs." in *NDSS*, 2017.

[101] E. Stefanov, M. v. Dijk, E. Shi, T.-H. H. Chan, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," *Journal of the ACM (JACM)*, vol. 65, no. 4, pp. 1–26, 2018.

[102] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing," in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: ACM, 2003, pp. 160–171. [Online]. Available: http://doi.acm.org/10.1145/782814.782838

[103] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, p. 437–450. [Online]. Available: https://doi.org/10.1145/2150976.2151022

[104] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX. New York, NY, USA: ACM, 2000, pp. 168–177.

[105] Tianchi, "Ad display/click data on taobao.com," 2018. [Online]. Available: https://tianchi.aliyun.com/dataset/dataDetail?dataId=56

[106] S. Tople, K. Grover, S. Shinde, R. Bhagwan, and R. Ramjee, "Privado: Practical and Secure DNN Inference," *CoRR*, vol. abs/1810.00602, 2018. [Online]. Available: http://arxiv.org/abs/1810.00602

[107] S. Tople, Y. Jia, and P. Saxena, "{PRO-ORAM}: Practical {Read-Only} oblivious {RAM}," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 197–211.

[108] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[109] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[110] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Brb: Mitigating branch predictor side-channels," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 466–477.

[111] R. Wang, Y. Zhang, and J. Yang, "D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 416–427.

[112] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2421–2434.

[113] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 850–861. [Online]. Available: https://doi.org/10.1145/2810103.2813634

[114] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[115] Y. Wang, E. Suh, W. Xiong, B. Lefaudeux, B. Knott, M. Annavaram, and H.-H. Lee, "Characterization of mpc-based private inferences for transformer-based models," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software*, 2022.

[116] Y. Wang, G. E. Suh, W. Xiong, B. Lefaudeux, B. Knott, M. Annavaram, and H.-H. S. Lee, "Characterization of mpc-based private inference for transformer-based models," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 187–197.

[117] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 494–505.

[118] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.

[119] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "{ScatterCache}: thwarting cache attacks via cache set randomization," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 675–692.

[120] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Huggingface's transformers: State-of-the-art natural language processing," 2019.

[121] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 640–656.

[122] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2003–2020.

[123] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. USA: IEEE Computer Society, 2003, p. 351.

[124] H. Yavarzadeh, A. Agarwal, M. Christman, C. Garman, D. Genkin, A. Kwong, D. Moghimi, D. Stefan, K. Taram, and D. Tullsen, "Pathfinder: High-resolution control-flow attacks exploiting the conditional branch predictor," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 770–784.

[125] C. Yin, B. Acun, C.-J. Wu, and X. Liu, "Tt-rec: Tensor train compression for deep learning recommendation models," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 448–462, 2021.
[126] Z. Zhang, T. Allen, F. Yao, X. Gao, and R. Ge, "Tunnels for bootlegging: Fully reverse-engineering gpu tlbs for challenging isolation guarantees of nvidia mig," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 960–974.
[127] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, "A lightweight isolation mechanism for secure branch predictors," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1267–1272.
[128] W. Zhao, J. Zhang, D. Xie, Y. Qian, R. Jia, and P. Li, "Aibox: Ctr prediction model training on a single node," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 319–328.
[129] Z. Zhao, L. Hong, L. Wei, J. Chen, A. Nath, S. Andrews, A. Kumthekar, M. Sathiamoorthy, X. Yi, and E. Chi, "Recommending what video to watch next: a multitask ranking system," in *Proceedings of the 13th ACM Conference on Recommender Systems*, 2019, pp. 43–51.
[130] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Untangle: A principled framework to design low-leakage, high-performance dynamic partitioning schemes," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 771–788.
[131] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Everywhere All at Once: Co-Location Attacks on Public Cloud FaaS," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2024. New York, NY, USA: Association for Computing Machinery, 2024.
[132] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2024. New York, NY, USA: Association for Computing Machinery, 2024.
[133] G. Zhou, N. Mou, Y. Fan, Q. Pi, W. Bian, C. Zhou, X. Zhu, and K. Gai, "Deep interest evolution network for click-through rate prediction," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 5941–5948.
[134] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai, "Deep interest network for click-through rate prediction," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 1059–1068.
[135] Y. Zhou, S. Wagh, P. Mittal, and D. Wentzlaff, "Camouflage: Memory traffic shaping to mitigate timing attacks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 337–348.
[136] Y. Zhou and D. Wentzlaff, "Mitts: Memory inter-arrival time traffic shaping," in *43rd International Symposium on Computer Architecture, ISCA 2016*. Institute of Electrical and Electronics Engineers Inc., 2016, pp. 532–544.

APPENDIX

A. Abstract

This artifact package includes training, profiling and inference scripts for the two deep learning applications with embedding tables discussed in the paper (DLRM and LLM), for various embedding generation techniques (non-secure table lookup, linear scan, ORAM, and DHE). The deep learning framework used is PyTorch, and all embedding generation techniques are built into PyTorch layers as extensions. The inference experiments are intended to be run on a CPU TEE (the training can be done faster on a GPU and is not the focus of the paper). Support is provided to run the CPU programs inside a TEE (Intel SGX on a 3rd Gen Intel Scalable Xeon Ice Lake Processor) via Gramine library-OS. The inference programs on CPU are to be investigated for performance (latency/throughput) and memory footprint. The training scripts are just to show that embedding generation via different techniques can reach comparable accuracies. Open-sourcing these implementations of the proposed embedding representations will allow other researchers to investigate and further adapt these solutions to possibly other and larger deep neural networks.

B. Artifact check-list (meta-information)

- **Algorithm:** embedding representations including table and Deep Hash Embedding (DHE); Deep Learning Recommendation Model (DLRM); recommendation systems; Large Language Model (LLM); natural language processing
- **Program:** Deep Learning models with modifications in embedding layer
- **Binary:** N/A, python-based, with libraries to be downloaded and custom C++ extensions to be compiled and installed
- **Model:** Deep Learning Recommendation Model (DLRM), originally from Meta. Large Language Model, namely GPT-2.
- **Data set:** for DLRM: Criteo Kaggle/Terabyte datasets; for LLM: OpenWebText corpus.
- **Run-time environment:** Linux OS, with SGX enabled
- **Hardware:** at least 3rd Gen (Ice Lake or later) Intel Scalable Xeon Processor with SGX enabled, no minimum core count but preferably 16
- **Run-time state:** not sensitive to cache state since focus is on data-oblivious algorithms
- **Execution:** sole system user; profiling
- **Metrics:** execution time; model accuracy; model memory footprint
- **Output:** execution timing information i.e. latency; training accuracy; model memory size
- **Experiments:** run provided python scripts and observe final output
- **How much disk space required (approximately)?:** datasets require 2TB, experiments require around 100GB
- **How much time is needed to prepare workflow (approximately)?:** multiple days to setup and preprocess datasets
- **How much time is needed to complete experiments (approximately)?:** multiple days to train, few days to run profiling and inference experiments
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT License
- **Data licenses (if publicly available)?:** Criteo Terabyte License (https://ailab.criteo.com/criteo-1tb-click-logs-dataset/), GPT-2 License (https://github.com/openai/gpt-2/blob/master/LICENSE)
- **Workflow automation framework used?:** N/A
- **Archived (provide DOI)?:** Uploaded to GitHub at https://github.com/bearhw/SecEmb_DHE and Zenodo at https://zenodo.org/records/14578272

C. Description

1) How to access: For the experiment scripts, clone repository from GitHub, at https://github.com/bearhw/SecEmb_DHE. The datasets have to be accessed separately. Pretrained models are available on Zenodo.

2) Hardware dependencies: This paper requires a CPU with a Trusted Execution Environment (TEE); we specifically target Intel CPUs with SGX. Hence, we require at least a 3rd Gen (Ice Lake or later) Intel Scalable Xeon Processor with SGX enabled and 64GB secure memory. Preferably, multiple cores are required, around 8-16, especially for LLM

experiments. Also, the CPUs should have AVX-512 SIMD support. For efficient training, GPUs are required, although slower training/fine-tuning can be done on a CPU.

3) Software dependencies: We run deep learning experiments via python-based framework i.e. PyTorch. The required packages can be most conveniently setup in an Anaconda environment www.anaconda.com. For SGX, the Gramine library OS (https://gramineproject.io/) needs to be installed.

4) Data sets: for DLRM: Criteo Kaggle/Terabyte datasets, publicly available and need to processed after downloading via instructions in our repo; these datasets also used in Meta's `dlrm` repo https://github.com/facebookresearch/dlrm; Require 2TB diskspace.

for LLM: OpenWebText https://huggingface.co/datasets/Skylion007/openwebtext; processing instructions in nanoGPT repo https://github.com/karpathy/nanoGPT. Require 50GB diskspace.

5) Models: Deep Learning Recommendation Model (DLRM), originally available at https://github.com/facebookresearch/dlrm;

Large Language Model originally available at https://huggingface.co/openai-community/gpt2-medium

D. Installation

To install Gramine on the system, follow the instructions at https://gramine.readthedocs.io/en/latest/installation.html, which will require root access. Also generate a Gramine key as per their instructions.

To setup Python and its various packages, first install Anaconda at www.anaconda.com. Then create a new environment `conda create -n pyth310 python=3.10`. Then install the following packages via `pip install package_name`: `pip install torch torchvision torchrec future numpy tqdm onnx pydot scikit-learn tensorboard shapely scipy matplotlib transformers datasets tiktoken wandb tqdm`.

Also install the assembler: `conda install anaconda::nasm`. Also install `libssl-dev` or equivalent.

Before running experiments, one also has to install our custom-implemented PyTorch extensions in the **`EXTENSIONS`** folder, using the `cmd-setup.sh` scripts.

E. Experiment workflow

There are two applications in our paper: DLRM and LLM. Each has its own directory in our repository, containing sub-directories for their training, embedding layer profiling and inference scripts.

F. Evaluation and expected results

The applications studied in our paper, DLRM and LLM, each have a top-level directory in our repository, containing sub-directories for their training, embedding layer profiling and inference scripts. The top-level directory contains a **`README`** file explaining the **detailed commands** to run for each stage.

DLRM

The training sub-folder can be used to launch training for table/DHE Uniform/DHE Varied, on the Criteo datasets, and also evaluate test accuracy of pretrained models. There are also scripts to convert DHE model to a table representation (for use in linear scan etc.).

The profiler/inference sub-folder has scripts to measure timing of embedding layers for various embedding generation techniques (non-secure, DHE Uniform, DHE Varied, Linear Scan, Path-ORAM, Circuit-ORAM), for various threads and batch sizes. Further, a Jupyter notebook is provided for thresholds calculation in the DLRM Hybrid scheme based on various system configuration. Additionally, the end-to-end inference scripts measure and output the overall model latency by employing various embedding generation techniques, including hybrid.

LLM

The training sub-folder provides scripts to finetune a GPT-2 medium model with table and DHE embedding layers on the OpenWebText dataset. One can also evaluate the loss on pretrained models, and sample the models for generation.

The profiling directory has scripts that allow measurement of various embedding generation techniques (Table/DHE/Linear Scan/Tree ORAM) for different batch and embedding table sizes. The inference directory allows end-to-end performance evaluation of the GPT-2 model when combined with various embedding generation techniques, by using a patched version of the HuggingFace `transformers` library.

MISC

In the `MISC` folder, we provide scripts to output the DLRM and LLM model memory footprints for various embedding generation techniques.