

TENNOR: Trustworthy Execution for Neural Networks through Obliviousness and Retrievals

Zifan Qu
George Mason University
Fairfax, VA, USA
zqu4@gmu.edu

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@cs.brown.edu

Giuseppe Ateniese
George Mason University
Fairfax, VA, USA
ateniese@gmu.edu

Evgenios M. Kornaropoulos
George Mason University
Fairfax, VA, USA
evgenios@gmu.edu

Abstract

Training neural networks on sensitive data in untrusted cloud environments requires simultaneously achieving computational efficiency and rigorous privacy guarantees. Sparsification techniques, essential for scalable training of wide layers, expose input-dependent memory-access patterns that are visible and can be exploited by a host OS/hypervisor, even when computation is protected by a Trusted Execution Environment. We present TENNOR, a system that resolves this tension by co-designing the neural network training pipeline with doubly oblivious primitives, eliminating value-dependent neural network access-pattern leakage while also utilizing adaptive sparsification. TENNOR recasts sparse neuron activation as a locality-sensitive hashing (LSH) retrieval problem, reducing secure sparsification to doubly oblivious accesses over an LSH data structure. To eliminate the prohibitive storage cost of "multi-table" LSH, we introduce Multi-Probe Winner-Take-All (MP-WTA): the first multi-probe scheme for rank-based LSH, achieving a 50× reduction in (hash table) memory while preserving model accuracy. We evaluate TENNOR on multi-label classification benchmarks, with output layers of up to hundreds of thousands of neurons under different batch sizes inside an Intel TDX Trusted Domain, achieving per-batch training speedups of 13×–470× over a Path ORAM baseline and reducing a 208-hour run to about 26 minutes.

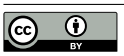
CCS Concepts

• **Security and privacy → Privacy-preserving protocols**.

Keywords

neural networks, obliviousness, trusted execution environments

ACM Reference Format:
Zifan Qu, Vasileios P. Kemerlis, Giuseppe Ateniese, and Evgenios M. Kornaropoulos. 2026. TENNOR: Trustworthy Execution for Neural Networks through Obliviousness and Retrievals. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3846750



This work is licensed under a Creative Commons Attribution 4.0 International License.

CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846750

1 Introduction

The success of neural networks (NN) has led the community to develop a range of architectures, each tailored to different tasks and presenting their own computational challenges. Whether the pipeline is built around convolutional networks for vision, transformers for language, or recommendation models for personalization [26, 39, 67, 89], the choice of architecture fundamentally shapes how computation must be organized, scheduled, and optimized. Among these, a particularly demanding class consists of networks containing *extremely wide layers*[1]: i.e., layers whose dimensionality reaches into the hundreds of thousands, or even millions, of neurons. Such layers arise naturally in the output layers of extreme multi-label classification and recommendation systems [6, 26, 59, 79, 98].

Efficient and Adaptive Access via Sparsification. Wide layers are computationally demanding even in standard, non-privacy-preserving settings. When a single layer contains tens of thousands of neurons, the cost of organizing, storing, and accessing its parameters becomes a bottleneck. This state of affairs has motivated a substantial body of work on *sparsification* [40]. That is, rather than activating all neurons in the wide layer for every input, only a small subset participates, dramatically reducing computation per access. Sparsification comes in two flavors: *non-adaptive* [32, 37], in which the network is pruned to permanently remove neurons that do not contribute to model performance; and *adaptive* [9, 80, 84], in which sparsity is applied dynamically, at access time, depending on the input. In this work, we focus on the latter. The resulting techniques for making wide-layer networks scalable have become a standard in modern systems [19, 31, 80].

While inference accesses the wide layer once, training requires repeated accesses. This repetition makes wide-layer training the dominant computational bottleneck, far exceeding the cost of inference in any realistic deployment scenario [19, 20, 79, 84]. Crucially, adaptive sparsification transforms wide-layer computation from dense matrix operations into sparse, irregular memory accesses, which is a workload that aligns poorly with GPU architectures that are optimized for regular data parallelism. CPUs, with their deep cache hierarchies and flexible control flow, are a natural fit for such access patterns, and *prior work has shown that sparse training on CPUs can match or outperform GPUs on wide-layer networks [20, 28].*

[1] The width of a network is defined to be the maximal number of nodes in a layer.

Trustworthy NN Training. In this work, we introduce a *trustworthiness* dimension to this setting by requiring that model parameters remain confidential via a Trusted Execution Environment (TEE), where an untrusted OS/hypervisor can observe memory access patterns even when data is encrypted. Unfortunately, sparsification, which is essential for scalability, becomes a liability in adversarial settings [45, 71]. The pattern of which a subset of neurons is adaptively activated for a given input is precisely the kind of access pattern that must not be revealed, since it leaks information about the input values. A growing body of attacks has demonstrated that *access patterns over encrypted data can be exploited* to recover the underlying plaintext [15, 35, 49–51]. Thus, naively applying sparsification to a wide layer exposes an attack vector, as memory accesses are revealed. This tension between (1) the efficiency gains from sparsification and (2) the access-pattern leakage motivates the privacy-preserving system we propose in this work.

TENNOR (which stands for Trustworthy Execution for Neural Networks through Obliviousness and Retrievals) is a system that is designed around CPUs and synthesizes *doubly-oblivious* primitives to enable private training on wide neural networks without revealing sparsification access patterns to the host OS/hypervisor. Central to our efficiency is the observation that sparsification can be cast as a retrieval task over a locality-sensitive hashing (LSH) data structure [19, 20, 43, 83, 84]. This framing allows us to adaptively identify active neurons without scanning the full layer, and to do so obliviously. We show that by co-designing the obliviousness mechanism with a sparse-activation structure, we achieve secure training on wide-layer networks at orders-of-magnitude lower cost than existing oblivious approaches, while providing rigorous guarantees against access-pattern leakage.

We make the following contributions:

(1) Doubly-oblivious Wide-layer Training Pipeline. We design and implement TENNOR, a doubly-oblivious training pipeline for wide neural networks within TEEs that eliminates value-dependent access-pattern leakage beyond the public input support. TENNOR carefully composes oblivious hash tables, oblivious sorting, and branch-free primitives into a provably secure adaptive sparsification pipeline, with security formally established via a simulation-based proof under a standard TEE adversarial model.

(2) Multi-Probe Winner-Take-All LSH. At the heart of our design is a newly introduced multi-probe scheme for rank-based LSH (dubbed MP-WTA), which defines a principled notion of neighboring buckets under WTA's ordinal hashing framework. MP-WTA achieves retrieval quality equivalent to that of 50 independent hash tables, but it uses only a single table, reducing LSH storage by 50× with negligible accuracy loss.

(3) Orders-of-magnitude Performance over Baseline. On extreme multi-label classification benchmarks, with hundreds of thousands of output labels, TENNOR achieves 13×–470× speedup over a Path ORAM baseline running inside Intel TDX. For this performance, we introduce three targeted optimizations (i.e., deferred dummy payload instantiation, parallel oblivious LSH scanning, and OHT layout reuse) that collectively eliminate the dominant bottlenecks while preserving double obliviousness.

2 Background

2.1 Trusted Execution Environments

Trusted Execution Environments (TEEs) (e.g., Intel SGX [24], ARM TrustZone [73], AMD SEV [1]) are *hardware-enforced, isolated runtime environments* that protect certain code and data from the rest of the system, including privileged software like the OS or a hypervisor. In VM-based TEEs, such as Intel TDX [22], the protected workload (including a guest OS, if present) runs inside a *Trust Domain* (TD), while the host and hypervisor remain outside the trust boundary. Beyond isolation, TEEs provide *attestation*, enabling remote parties to verify that a workload is genuinely executing within a secure enclave. Together, these guarantees underpin cloud *confidential computing*, such as secure machine learning (ML) services, allowing users to outsource sensitive data and computations without exposing their contents to an infrastructure provider.

Despite these protections, TEEs do not defend against most classes of *side-channel attacks*. Prior work has demonstrated attacks compromising *confidentiality* by exploiting leakage from secret-dependent memory accesses or control-flow behavior [12, 52, 53, 62, 63, 82, 88, 96]. Complementing architectural [11, 25] and software defenses [21, 81], *data-oblivious programming* offers an alternative mitigation strategy: programs are written such that their memory access patterns are independent of any secret inputs (§3).

2.2 Oblivious Computation

At a high level, a computation is *oblivious* if its memory access pattern depends only on public information (e.g., input length and fixed parameters) and not on sensitive values. More formally, a computation is oblivious iff any two input sequences of the same length induce memory access patterns that are indistinguishable to a computationally-bounded adversary. Obliviousness further requires that branches, loop bounds, and memory indices in the executed code do not depend on secret data.

Oblivious RAM. *Oblivious RAM* (ORAM), introduced by Goldreich and Ostrovsky [33], is a general-purpose technique for achieving obliviousness. At a high level, ORAM conceals which logical block is accessed by performing *additional* reads and writes and periodically shuffling data, so that the observed access pattern reveals only the number of operations, not their targets. ORAM has been substantially improved over the past few decades [4, 5, 72, 85, 92], yet it remains a generic construction. *Application-specific oblivious designs have been shown to achieve significantly better performance, as we also demonstrate in this work.*

Application-tailored Obliviousness. Beyond generic ORAM, the research community has developed *oblivious data structures* (ODS) [93] and oblivious algorithms tailored to specific tasks. Prior work has produced oblivious counterparts for a range of algorithms and data structures, including transactional and map-like systems [27, 86], search indexes and database query processing [30, 60], hashing [17], and sorting [3, 8, 69, 78]. At the primitive level, basic operations such as *comparison*, *swap*, and *copy* must also be oblivious. Several works [29, 69, 77, 78, 86] employ branch-free primitives, such as CMOV (x86) instructions and bitwise masking, to ensure that access patterns are independent of the input.

Double Obliviousness. In TEEs, access-pattern leakage can arise from three sources: (1) memory accesses inside the TEE, (2) accesses to encrypted memory outside the TEE, and (3) contention on shared microarchitectural resources. Early work largely focused on concealing access patterns at the boundary between the TEE and external memory—i.e., point (2). This scope was broadened by Oblix [60], which introduced the notion of *double obliviousness*, requiring that access patterns within the TEE are also protected—i.e., point (1). Subsequent works [16, 29, 86] adopt this definition, ensuring that operations inside the TEE are themselves oblivious.

2.3 CPU-friendly Neural Networks

Deep neural networks (DNNs) are commonly trained on GPUs and specialized accelerators, as dense layers map naturally to large matrix multiplications. In this work, we target CPU-based TEEs, making CPU-efficient DNN designs particularly relevant; more broadly, *CPUs are the platform of choice when GPUs are unavailable (e.g., at the edge) or when power and cost constraints dominate* [61].

Running DNN inference on general-purpose CPUs is challenging for two reasons. First, arithmetic throughput is limited: CPUs have fewer cores and narrower SIMD units, making naive dense-layer execution orders of magnitude slower. Second, performance is often memory-bound: CPUs rely heavily on caching to compensate for slower RAM access, so models whose working sets exceed the cache, or whose access patterns are irregular, suffer significant slowdowns.

Sub-Linear Deep Learning Engine. Chen et al. [20] introduced the *Sub-Linear Deep Learning Engine* (SLIDE): a CPU-oriented framework for training and inference that replaces dense computation with adaptive sparsity. Given an input, SLIDE uses *locality-sensitive hashing* (LSH) to identify a small, input-dependent active subset of neurons—in wide layers—and restricts forward and backward passes to this subset. This reduces dense matrix multiplication to a small number of hash-table lookups and sparse dot products, better exploiting CPU strengths. However, because the active set is only an approximation of the true top-activated neurons, accuracy can degrade if the active set is too small. More critically for our setting, *the accessed hash buckets and updated neurons depend directly on the input, making naive SLIDE-style execution highly access-pattern-sensitive and therefore insecure inside a TEE.*

2.4 Locality-sensitive Hashing

Locality-sensitive hashing (LSH) is a family of randomized hash functions with the property that similar items are more likely to *collide* (i.e., land in the same bucket) than dissimilar ones [43]. Unlike traditional hashing, collisions in LSH are intentional: they provide a fast mechanism for retrieving a short list of candidates without scanning the full dataset.

Several LSH families have been proposed for different similarity measures, including MinHash for Jaccard similarity [13], SimHash for cosine similarity [18], and Winner-Take-All hashing for rank-based comparisons [97]. LSH has found application across a range of domains, including approximate nearest-neighbor (ANN) search [2], recommendation systems [83], and accelerating ML [47]. *SLIDE uses LSH for fast network sparsification during training* (§2.3).

More formally, let d be the *distance* measure on the domain S.

Definition 1 (LSH family). *A function family H is called $(r, c \cdot r, p_1, p_2)$-sensitive for d, if for any $p, q \in S$:*

- *If $d(p,q) \le r$, then $\Pr_{h \leftarrow H}[h(p) = h(q)] \ge p_1$,*
- *If $d(p,q) > c \cdot r$, then $\Pr_{h \leftarrow H}[h(p) = h(q)] \le p_2$*

For ANN search, we require $c > 1$ and $p_1 > p_2$: points within distance r collide with probability at least p_1, while points farther than cr collide with probability at most p_2. In practice, this gap is amplified by concatenating multiple hash functions and/or maintaining multiple hash tables, then probing the corresponding buckets to retrieve candidates. We next describe Winner-Take-All hashing, the rank-based LSH family used by both SLIDE and TENNOR.

2.5 Winner-Take-All Hashing

Winner-Take-All (WTA) [97] is an LSH family for rank-based similarity that captures *ordinal relationships* between features rather than metric distances. Unlike traditional LSH, WTA is invariant to positive rescaling of features, making it particularly suitable for high-dimensional representations common in deep learning.

LSH for Rank Correlation. Most LSH families target metric distances, such as Euclidean or angular distance. WTA instead targets *rank correlation*, which depends only on the relative ordering of feature values rather than their absolute magnitudes. Under this notion, two vectors are similar when their features share the same ranking, even if the actual values differ. Rank correlation is quantified by counting pairwise order agreements between two rankings as:

$$PO(X,Y) = \sum_{i=1}^{n} \sum_{j<i} T\left((x_i - x_j)(y_i - y_j)\right), \tag{1}$$

x_i and y_i are the values of the i^{th} feature of $X, Y \in \mathbb{R}^n$ and $T(z)$ is a threshold function that takes value 1, if $z > 0$, or value 0, if $z \le 0$. Informally, this counts the number of feature pairs on which X and Y agree in ordering (i.e., either both $x_i > x_j$ and $y_i > y_j$, or vice versa). A high PO value indicates that the two vectors induce very similar rankings over their n features. *Rank correlation is particularly well-suited for domains with noisy, heterogeneously scaled, or sparse feature representations.*

WTA Hashing. While rank correlation is theoretically well-founded, exact computation requires $O(n^2)$ comparisons per vector and does not scale. WTA approximates rank similarity by sampling a random subset of features and recording only the winner—i.e., the index of the maximum value—within that subset. Concretely, each hash function $h_i(\cdot)$ is parameterized by a fixed random permutation and a window size M; given a vector X, it outputs the index of the maximum among the M sampled features. Applying K such functions yields a *signature* $\mathbf{h}(X) = (h_1(X), \ldots, h_K(X))$. *Two vectors with similar rankings tend to agree on the winner across many sampled windows, so the Hamming distance between their signatures serves as a proxy for rank-based similarity.*

3 Threat Model

We assume a TEE-based threat model, aligned with prior work [16, 60, 86], which *combines protected execution with double obliviousness.* In our prototype, the TEE is instantiated as an Intel TDX TD/VM.

We assume a *powerful adversary* $\mathcal{A}$ that: (1) controls the host software stack (e.g., the host OS and/or hypervisor); and (2) can observe memory access patterns generated by the TD during execution.

In particular, $\mathcal{A}$ can observe memory accesses at both page and cache-line granularity. However, $\mathcal{A}$ cannot access processor-internal secrets within the TEE, including the hardware root key (used for memory encryption, attestation, key derivation, etc.), nor can it inspect CPU register contents during TEE execution. *Our guarantee applies at the algorithmic access pattern level.*

Additionally, we consider power-analysis [48, 55], rollback [57, 99], timing-based [94, 95], and denial-of-service [22] attacks out of scope. We use Snoopy's [29] code primitives (i.e., copy, sort), but we do not claim whole-program, binary-level obliviousness after compiler optimizations. Our goal is to eliminate value-dependent NN access-pattern leakage under the aforementioned adversarial observation capabilities.

Our system runs inside a Trusted Domain (TD) provided by Intel TDX on an untrusted cloud server. The user's client establishes a secure channel to the remote TD/TEE using remote attestation, derives a session key, and transmits sensitive data over this channel. The code executes within the TD, and all sensitive data are processed within the TD. Unlike an SGX-style enclave setting, where one often distinguishes enclave memory from memory *outside* the enclave, the relevant boundary here is the TD/VM vs. entities *external* to the TD/VM (i.e., OS, VMM). Although our prototype focuses on Intel TDX, the threat model considered relies only on standard TEE guarantees (e.g., protected execution, memory encryption) and therefore the same high-level objectives are also relevant to other TEEs, such as Intel SGX and AMD SEV—though the concrete trust boundary and system instantiation may differ.

End-to-end Obliviousness. TENNOR treats the number and positions of non-zero input features as public parameters, while the corresponding feature values remain private. Specifically, TENNOR protects value-dependent access patterns within the neural network but does not hide which input features are non-zero. The reported measurements in this work exclude the additional cost of hiding which input features are non-zero, but we discuss a potential front-end that can make TENNOR end-to-end oblivious.

One straightforward way to create a front-end that hides which input features are non-zero is to "touch" and compute on every dimension in the original input, including dimensions whose values are zero. Towards analyzing the complexity of this approach, let D denote the input dimensions and N_0 the number of neurons in the first layer. For each batch of B input vectors, the front end scans all D dimensions in a fixed order and performs the same multiply-accumulate operations for each of the N_0 first-layer neurons, regardless of the input value. This task can be parallelized by assigning fixed ranges of features or neurons to p threads, resulting in $O(BN_0D)$ total work parallelized in $O(\frac{BN_0D}{p})$ time. Across all workloads evaluated in Section 7.1, this front-end adds at most 0.07% to TENNOR's per-batch runtime for 44 threads. Note that this overhead is small because the sequential scan of high-dimensional inputs exhibits strong memory locality. In terms of computation, every input feature needs to perform a multiplication and a write; thus, the end result is doubly oblivious. The extended version [74] of our manuscript provides the per-dataset measurements.

4 Overview

4.1 Key Insights

Secure Neural Network training on an untrusted cloud requires *double obliviousness* at the algorithmic access-pattern level, i.e., data-dependent access patterns, across the memory hierarchy, must be avoided. The key challenge is to introduce new techniques and carefully compose existing ones, tailored to the wide NN training pipeline so as to produce an efficient doubly-oblivious system.

❶ **NN-adaptive Sparsification via Data Structure Retrieval.** Textbook NN training processes all neurons for each input. More advanced approaches apply static pruning to sparsify the network, reducing both computation and storage. Instead, TENNOR *dynamically selects a small subset of dominant neurons, at runtime, via adaptive sparsification.* Specifically, TENNOR transforms dense matrix multiplication into an LSH table-retrieval task, a method originally introduced by Chen et al. [20]. By casting sparsification as a retrieval problem, TENNOR avoids computing the subset of activated neurons via MPC or HE. Instead, TENNOR performs lookups in an LSH table that groups neurons during initialization to facilitate sparsification. *Consequently, secure sparsification reduces to performing doubly-oblivious accesses to the LSH table(s).*

❷ **Reduced LSH Tables via Multi-Probe WTA.** Given that LSH is a data structure with probabilistic guarantees, a standard approach to amplify the probability of success is to construct multiple LSH tables and fetch a bucket from each. Instead, TENNOR *introduces a novel LSH scheme for WTA that retrieves multiple buckets from a single table*, rather than retrieving a single bucket from multiple tables (as SLIDE [20] does). This *Multi-Probe WTA* scheme, which is of independent interest, preserves NN training accuracy (see §7.3) while reducing space usage by 50×.

❸ **Tailored Doubly-oblivious Design.** With these design choices, wide NN training within TENNOR reduces adaptive sparsification to mapping training data points to neurons. The key efficiency question is which collection of "objects" (e.g., training data points or neurons) should serve as the target of oblivious fetching. Rather than fixing a training point and obliviously scanning across all neurons, TENNOR *reverses this approach completely: for each fixed neuron, it obliviously fetches the training points that require it.* Naively applying this idea to the entire training set would be costly, due to the scalability limitations of oblivious computation over such a large collection (of training data). To address this, TENNOR partitions the training data into batches and performs oblivious fetching per batch. Overall, the linear scan of the LSH table that stores neurons is doubly-oblivious, while the request entries within each batch are organized in a doubly-oblivious hash table [17], providing the overall obliviousness guarantees. Hence, under cache-line-level leakage, TENNOR provides double obliviousness.

4.2 Approach

Figure 1 shows the high-level workflow and components of TENNOR. The system operates entirely within an Intel TDX Trusted Domain (TD) and is designed to provide double obliviousness.

NN Preprocessing. During initialization, TENNOR samples hash functions for the wide layer of the NN and stores that layer in the LSH key-value store shown on the right-hand side of Figure 1.

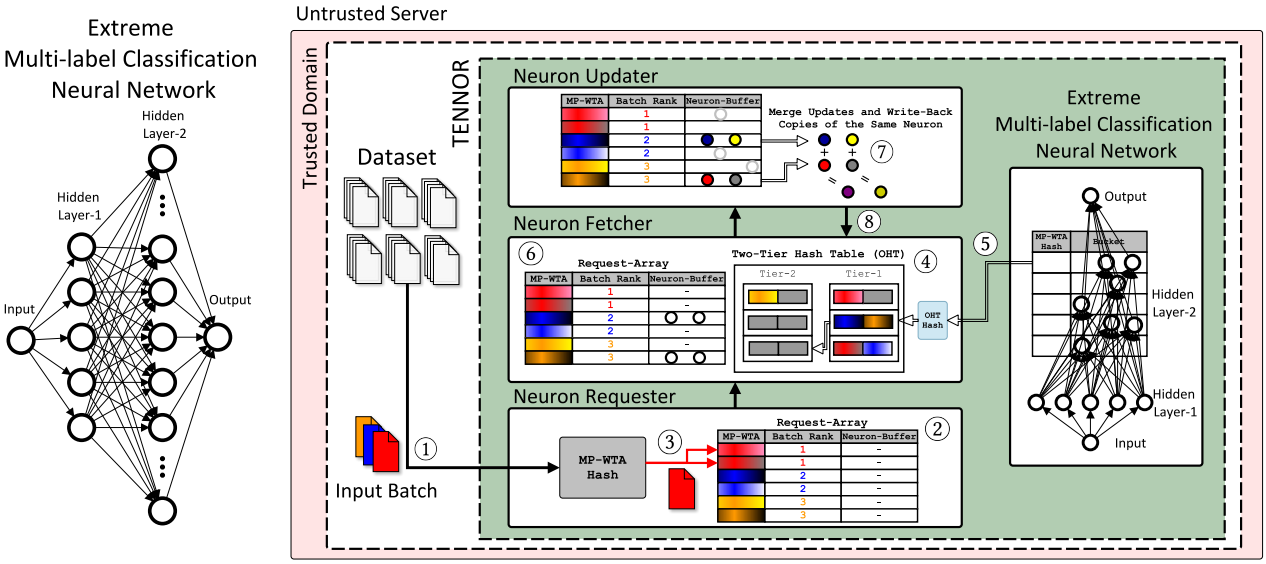


Figure 1: Design of TENNOR. The components `Neuron Requester`, `Neuron Fetcher`, and `Neuron Updater` follow a doubly-oblivious design when accessing or updating the wide NN stored in Locality-Sensitive Hashing (LSH) table(s), shown on the right-hand side. The LSH tables use our newly introduced Multi-Probe Winner-Take-All (MP-WTA) hashing scheme.

Importantly, each LSH key-value pair stores the full information associated with a neuron (e.g., weights, gradients, bias), rather than a pointer that must be dereferenced to retrieve neuron state from another data structure. The training data is then partitioned into batches. ① The workflow begins when a batch of training inputs is transmitted to TENNOR, and execution proceeds through three doubly-oblivious components (outlined below).

Neuron Requester. Each input in the batch "requests" a subset of active neurons for training. Multiple inputs may require the same neuron, which can lead to redundant fetches. ② To handle this efficiently, we organize neuron requests from all inputs into a `Request-Array` that stores the triplet: (*i*) the probed `LSHbucketID`, (*ii*) the input's position in the batch, and (*iii*) a placeholder buffer for the neuron data (to be populated later). ③ A single training data point probes multiple locations in the LSH table, generating multiple entries in the `Request-Array`.

Neuron Fetcher. The `Request-Array` does not group requests by neuron, so determining which training data points in a batch trigger a particular neuron requires additional bookkeeping. ④ To address this, TENNOR transforms the `Request-Array` into an Oblivious Hash Table (OHT) [17], which acts as an oblivious key-value store where the key is an `LSHbucketID` and the value consists of the corresponding `Request-Array` triplet(s). At this stage, the triplets do not yet store any neurons. ⑤ Next, TENNOR performs a linear scan over the LSH table; at each iteration, the current `LSHbucketID` is used as a key to the OHT to locate the associated triplet(s) and populate them with the neurons stored in the corresponding bucket. ⑥ After the scan completes, the OHT is transformed back into a *populated* `Request-Array` containing the requested neurons. If multiple triplets request the same `LSHbucketID`, TENNOR creates a separate copy of the padded bucket contents for each triplet.

Neuron Updater. Once the `Request-Array` has been populated, TENNOR performs feedforward and backpropagation only on the neurons contained in it. ⑦ Because the same neuron may appear in multiple triplets, TENNOR obliviously aggregates the gradient updates from all copies to compute a single consolidated update as part of the backpropagation. ⑧ Finally, TENNOR obliviously writes the updated neurons back to the LSH table by invoking Neuron Fetcher again, this time with the updated neurons. Unlike the first invocation of Neuron Fetcher (which takes place during feedforward) that transfers neurons from the LSH table to the OHT, this step (now in backpropagation) writes the updated neurons from the OHT back to the LSH table.

Although our prototype targets Intel TDX, its modular design allows it to be adapted to other TEE implementations, such as Intel SGX and AMD SEV, with minimal (if any) changes.

5 Design and Implementation of TENNOR

5.1 LSH Storage Reduction

Recall that amplifying the probability of success in LSH typically requires constructing multiple hash tables and querying the corresponding bucket in each table. An alternative approach from the information retrieval community, called *Multi-Probe LSH* (MP-LSH) [56], addresses this requirement by probing multiple buckets within a *single* hash table. The key intuition is that if a stored item is "close" to the query item q, but does not fall into the same bucket, it is likely to reside in a "nearby" bucket—i.e., a bucket whose hash value differs only slightly from that of q. Accordingly, MP-LSH probes a sequence Seq of neighboring buckets around the original hash $\mathbf{h}(q)$ *within the same table*: $\mathsf{Seq}(q) = (\mathbf{h}(q), \mathbf{h}_1(q), \mathbf{h}_2(q), \ldots, \mathbf{h}_z(q))$.

Each bucket $\mathbf{h}_i(q)$ differs from the original $\mathbf{h}(q)$ by a *perturbation* δ_i. This strategy achieves search quality comparable to standard LSH while requiring significantly fewer hash tables.

Multi-Probe WTA. To the best of our knowledge, no multi-probe scheme has been proposed for WTA in the literature. The main challenge lies in defining what constitutes a "neighboring" bucket within WTA's rank-based hashing framework. In the original MP-LSH [56] work, perturbations are typically ±1. That is, for a given dimension of $\mathbf{h}(q)$, the two neighboring probe buckets are obtained by incrementing or decrementing that dimension's integer value by one. In contrast, WTA hashing operates differently; it samples a subset of elements from the query vector and returns the index of the maximum element within that subset. Consequently, unlike LSH for Euclidean distance, there is no obvious notion of a "neighbor" LSH bucket in this ranking-based setting.

Our Approach. Suppose the WTA signature of a query (or vector) $q \in \mathbb{R}^m$ is $\mathbf{h}(q) = \big(h_1(q), \ldots, h_K(q)\big)$, where each hash $h_i(q)$ returns the index of the maximum feature value among the M sampled features used in the i^{th} WTA function. If multiple sampled features have the same value, ties are broken deterministically according to the fixed sampled order of that hash function. Our approach defines a neighboring LSH bucket by considering the index of the *second-largest* feature value within the corresponding sampled subset; this index is denoted by $h_i^2(\cdot)$. We generate a perturbed hash by selecting a signature dimension i and replacing $h_i(q)$ with $h_i^2(q)$. Consider two vectors with nearly identical feature values except for a single feature that appears as a large outlier in one of the vectors; if this outlier is sampled during the computation of h_i, its index will dominate that position of the signature. *By falling back to the second-largest value, we obtain a more robust representation that avoids penalizing otherwise similar vectors that differ only in the identity of the maximum feature.*

Further Relaxation. The above approach perturbs a single position of $\mathbf{h}(q)$ using one specific alternative. We extend this idea along two axes. First, instead of considering only the second-largest value, we allow the perturbed position to use the index of the r-th largest feature value, where $2 \le r \le M$. We define $h_i^r(q)$ as the function that returns the index of the r-th largest element within the sampled subset associated with the i-th WTA hash function.

Second, we allow simultaneous perturbations of up to N positions of signature $\mathbf{h}(q)$, where $1 \le N \le K$. If the number of perturbed positions is i, where $1 \le i \le N$, then there are $\binom{K}{i}$ ways to choose the i perturbed positions among the K signature dimensions. For each chosen position, there are $(r-1)$ alternative hash values available, namely from the second-largest $h_*^2(q)$ through the r-th largest $h_*^r(q)$. Thus, there are $(r-1)^i$ possible assignments of alternative hash values across the i perturbed positions.

Given the above, the total length of the probing sequence is $\mathsf{lenSeq} = 1 + \sum_{i=1}^{N} \binom{K}{i}(r-1)^i$, where the first term corresponds to the original LSH bucket $\mathbf{h}(q)$. In our implementation, we set $r = 3$ and $N = 3$. Under these parameters, position i of the signature can be replaced by either h_i^2 or h_i^3, and up to three positions may be perturbed simultaneously. Algorithm 1 describes our `MP-WTA`.

Note that the parameters are fixed in advance, independently of the training data, yielding an oblivious multi-probe schedule.

Algorithm 1 MP-WTA$_{r=3,\,N=3}$

Require: Query vector q, Sequence of WTA hash functions $h_1, \ldots, h_K$
Ensure: Probing sequence Seq of length $\mathsf{lenSeq} = 1 + \sum_{i=1}^{N} \binom{K}{i}(r-1)^i$, with $r = 3$ and $N = 3$

```
Compute h(q) = (h_1(q), ..., h_K(q)) and Seq ← ∅
for i ∈ {1, ..., K} do compute h_i^2(q), h_i^3(q) ▷ 2nd and 3rd maximum for all h_i
Seq ← (h(q))
for n ∈ {1, 2, 3} do                                   ▷ number of positions to perturb
    for each subset S ⊆ {1, ..., K} of positions s.t. |S| = n do
        for each tuple (r_j)_{j∈S} with values r_j ∈ {2, 3} do
            h̃ ← h(q)                                   ▷ copy original hash as the base for perturbation
            for j ∈ S do h̃_j ← h_j^{r_j}(q)            ▷ replace with 2nd or 3rd maximum
            Append h̃ to Seq
return Seq
```

The full exposition of $\mathbf{h}(\cdot), h_*^2(\cdot), h_*^3(\cdot)$'s oblivious algorithms is available in the extended version [74] of our manuscript.

5.2 TENNOR Components

Before describing the components, we fix the public parameters used by `TENNOR`. Let P denote the collection of public parameters known to the adversary. These include the batch size, the NN architecture and layer dimensions, the layer(s) to which LSH is applied, the public LSH parameters, the number of LSH buckets, the public probe-sequence length, the public padding parameter `PADSIZE`, the public parameters of the `OHT`, such as the number of bins and the bin capacity, the LSH rebuild period, the optimizer hyperparameters, and the current training-step index. For the first layer, we also treat the non-zero feature indices of the input as public. We treat the public number of training steps or batch iterations as part of P. These parameters determine the shapes of intermediate arrays, the bounds of all loops, and the schedule of memory traversals used by the components. All other information is private, including the values of the training data, the values of the model parameters, optimizer state, intermediate activations, gradients, and the concrete mapping from requests to buckets and from buckets to fetched neurons. (Refer to our extended manuscript [74] for additional details.)

5.2.1 Neuron Requester. In our prototype, `TENNOR` applies LSH-based adaptive sparsification only to the wide layer, which is wide by nature in the extreme multilabel classification setting and thus dominates the training cost (see §4). We note that other parameterizations of `TENNOR` may apply LSH to multiple (even all) layers. The term *input* to the layer being processed by the Neuron Requester refers to the activation vector fed into that layer. Algorithm 2 processes the batch's fetch requests for LSH that correspond to NN layer ℓ and creates an intermediate data structure `Request-Array`. For a batch of B inputs, Neuron Requester invokes `MP-WTA` on each input q_i to obtain a probe sequence Seq_i of length lenSeq_ℓ.

Obliviousness. To preserve obliviousness, the Neuron Requester uses a statically pre-allocated array $\mathsf{ReqArray}_\ell$ of $T = B \cdot \mathsf{lenSeq}_\ell$ entries instead of variable-sized data structures, whose dynamic resizing pattern would leak information about the data. The probe/request with local index t for input i is placed in the $(i-1) \cdot \mathsf{lenSeq}_\ell + t$ slot of the `Request-Array`. Each entry stores a triplet consisting of (*i*) the signature `LSHbucketID`, (*ii*) `Batch-Rank`, and (*iii*) a `Neuron-Buffer` containing `PADSIZE` dummies.

Algorithm 2 Neuron Requester for layer ℓ

Require: Layer index ℓ, batch of input data of size B, hash functions $h_1, \ldots, h_K$, and the maximum number of neurons for each `Neuron-Buffer`, denoted by PADSIZE
Ensure: Initialized ReqArray_ℓ
1: $K \leftarrow$ # of hash functions at layer ℓ
2: $\text{lenSeq}_\ell \leftarrow$ length of probing sequence `Seq` for the LSH of layer ℓ
3: $T \leftarrow B \cdot \text{lenSeq}_\ell$ ▷ total number of LSH probes for batch B
4: Allocate empty `Request-Array` $\text{ReqArray}_\ell[0..T-1]$
5: *// Run MP–WTA and construct* ReqArray
6: **for** each input $i = 1$ to B **do**
7: $\text{Seq}_i \leftarrow \text{MP-WTA}_{r=3,\,N=3}(q_i, (h_1, \ldots, h_K))$, where q_i is an entry of the batch
8: **for** $t = 0$ to $\text{lenSeq}_\ell - 1$ **do**
9: Index of ReqArray to populate is $\texttt{ind} \leftarrow (i-1) \cdot \text{lenSeq}_\ell + t$
10: $\text{ReqArray}_\ell[\texttt{ind}].\texttt{LSHbucketID} \leftarrow \text{Seq}_i[t]$
11: $\text{ReqArray}_\ell[\texttt{ind}].\texttt{Batch-Rank} \leftarrow i$
12: $\text{ReqArray}_\ell[\texttt{ind}].\texttt{Neuron-Buffer} \leftarrow$initialize with PADSIZE dummies
13: **return** ReqArray_ℓ

Algorithm 3 Neuron Fetcher for layer ℓ (Read mode; Write mode reverses the transfer direction)

Require: ReqArray_ℓ of $T = B \cdot \text{lenSeq}$ entries; LSH table LSH_ℓ with numLshBucs buckets, each padded to PADSIZE; public OHT parameters numBin, binCap
Ensure: each OHT entry's `Neuron-Buffer` populated with its corresponding LSH bucket's neurons
1: **// Phase 1: build OHT from Request-Array, keyed by LSHbucketID**
2: OHT.BUILD(ReqArray_ℓ, numBin, binCap)
3: **// Phase 2: linear scan over all LSH buckets**
4: **for** $b = 0$ to numLshBucs $- 1$ **do**
5: **for** tier $\in \{1, 2\}$ **do**
6: **for** each of the binCap entries e in bin OHT.LOOKUP($tier, b$) **do**
7: $match \leftarrow$ OBLCOMPARE(e.LSHbucketID, b)
8: e.Neuron-Buffer $\leftarrow$ OBLCHOOSE($match$, $\text{LSH}_\ell[b]$, e.Neuron-Buffer)
9: **// Phase 3: restore fixed-length Request-Array**
10: collect all OHT entries into flat array A
11: OBLSORT(A, key = (isDummy, Batch-Rank))
12: **return** $A[0..T-1]$ ▷ populated ReqArray_ℓ

This way `Request-Array` has a fixed structure for the subsequent fetch stage, and its layout does not leak information about the input. Although the LSH bucket signature itself depends on the input values, Algorithm 1 computes the signature in a data-oblivious manner. We note that public system parameters P fully determine the observable quantities such as the size of ReqArray_ℓ, the number of probes issued per input, and the write schedule. Thus, the observable layout and write schedule of the Neuron Requester reveal nothing beyond what is already known via P. Notice that lines 6–12 of Algorithm 2 can be further optimized by populating each entry of the `Request-Array` in parallel since there is no dependence across the content or the computation of `Request-Array`'s entries.

5.2.2 Neuron Fetcher. The main design goal of Neuron Fetcher is to populate the `Neuron-Buffer` for each `Request-Array` entry. To do that, buckets from the LSH table need to be retrieved and copied to the memory of `Neuron-Buffer`. Specifically, Neuron Fetcher's retrieval step must hide two data-dependent pieces of information: (1) which LSH buckets are accessed during retrieval, and (2) the number of neurons residing in each queried bucket for a given request. TENNOR addresses the latter by padding every LSH bucket to a fixed public PADSIZE capacity with dummy neurons, so that each bucket retrieval fetches the same number of neurons. Hiding which buckets are accessed requires a different mechanism.

As described in Section 4.2, for efficiency reasons, TENNOR adopts a reverse-mapping strategy; it iterates over LSH buckets and for each of them it identifies all requests that target it. To achieve the desired security guarantees, we need a data structure that can *obliviously* map LSH signatures to requests.This is achieved by reorganizing ReqArray_ℓ into a two-tier Oblivious Hash Table (`OHT`), as proposed by Chan et al. [17]. `OHT` stores one entry per request. The key is `LSHbucketID`; the value is a triplet of `LSHbucketID`, `Batch-Rank`, and `Neuron-Buffer`. Thus, multiple requests in the batch may appear as distinct `OHT` entries with the same `LSHbucketID`.

Because the `OHT` maps keys to bins using a hash function, entries (requests) with different `LSHbucketID`s may be placed in the same bin due to hash collisions. To allow the fetcher to identify the correct entries when scanning a bin, each stored triplet also records its `LSHbucketID` as a field, in addition to the `OHT` lookup key. In more detail (see Algorithm 3), Neuron Fetcher linearly scans all LSH buckets, computes the corresponding tier-1 and -2 `OHT` bins for the current bucket, and linearly inspects the entries in those bins.

For each candidate entry, Neuron Fetcher uses: `OblCompare` [29], a *data-oblivious comparison primitive*, to test whether its `LSHbucketID` matches the current LSH bucket; and `OblChoose` [29], a *data-oblivious conditional selection+assignment primitive*, to copy that bucket's padded contents into the corresponding `Neuron-Buffer`(s). (See our extended manuscript [74] for details.) After all buckets have been processed, Neuron Fetcher obliviously transforms the `OHT` back to `Request-Array`, whose `Neuron-Buffers` now hold the retrieved neuron contents. The full version details this procedure. The design of Neuron Fetcher is *doubly-oblivious*. To build the `OHT`, we instantiate `OblSort` with Batcher's bitonic sorting network, whose compare and exchange schedule is fixed and data-independent [8].

Obliviousness. Algorithm 3 summarizes the Fetcher's fixed three-phase schedule: oblivious OHT construction, a complete scan of all LSH buckets and corresponding fixed-capacity OHT bins, and oblivious recovery of the `Request-Array`. As established by Chan et al. [17], double obliviousness requires that each key of the `OHT` be queried at most once. This requirement is satisfied by our construction; since each padded LSH bucket is scanned exactly once during the linear pass, no key is ever accessed more than once. Finally, to convert the `OHT` back into a *populated* `Request-Array`, we again apply oblivious sorting.

Optimization #1 (O1): Deferred Dummy Payload. Recall that during the feedforward phase, the `Request-Array` must be transformed into an `OHT` via an oblivious sort operation. At this point, however, the `Neuron-Buffer` field of each `Request-Array`'s triplet is occupied *solely by dummy values*. Moving these placeholders through the sort operation, i.e., rewriting them alongside the meaningful fields of the triplet, introduces significant overhead (see §7.2) without any security benefit.

Obliviousness. TENNOR optimizes performance based on the above observation by *deferring dummy payload instantiation*; during construction, the oblivious sort operates only on the fields `LSHbucketID` and `Batch-Rank` of `Request-Array`. Once all request items have been placed into their target `OHT` bins, TENNOR initializes every `Neuron-Buffer` with dummy neurons, thereby restoring the fixed-capacity layout required by the subsequent fetch stage. Both the oblivious sorting and the subsequent dummy-initialization process operate on a fixed number of entries in a fixed order.

Thus, this optimization preserves obliviousness. TENNOR avoids redundant oblivious movement of dummy Neuron-Buffers without affecting the obliviousness of feedforward OHT construction.

Optimization #2 (O2): Oblivious Parallelization. Our experiments in Section 7.2 show that the linear scan of the LSH table that populates the OHT is a performance bottleneck. In this design, each LSH bucket encountered during the scan immediately triggers its corresponding Tier-1 and Tier-2 OHT lookups, forcing LSH bucket iteration and OHT updates to proceed in lockstep. TENNOR breaks this lockstep by deferring all OHT updates to a separate phase after the scan completes, enabling parallel updates across threads.

Naively parallelizing across LSH buckets would introduce write contention, since multiple buckets may want to write to the same OHT bin. TENNOR eliminates this contention through a two-phase approach. In the first phase, TENNOR iterates over the LSH buckets and initializes a *scheduler* (in fact, one per OHT tier) that records which LSH buckets map to which bins (as determined by the OHT hash functions). In the second phase, once the LSH iteration completes, TENNOR partitions the OHT bins into disjoint regions and assigns each region together with its associated LSH buckets to a dedicated thread, guaranteeing that no two threads write to the same bin.

Obliviousness. This optimization preserves obliviousness. The scheduler is derived from the fixed sequence of LSH bucket indices and the OHT hash functions, so its construction has no data-dependent leakage. The resulting partition assigns OHT-bin regions to threads without leakage, beyond the public parameters, and does not alter the set of buckets, bins, or entries processed. Notice that this parallel Neuron Fetcher performs *the same* LSH table iteration, updates *the same* OHT bins, and applies *the same* oblivious operations as the sequential design. The only difference is the order of OHT updates, not which memory locations are accessed.

Optimization #3 (O3): OHT Layout Reuse. Both feedforward and backpropagation construct an OHT from scratch over a Request-Array, each incurring multiple rounds of oblivious sorting that our experiments identify as another bottleneck (see §7.2). Since the OHT construction in both phases operates on the same set of requests, the request-to-OHT-bin assignment in OHT established during feedforward remains valid for backpropagation and need not be recomputed. To support reuse, during the feedforward OHT construction, TENNOR records each request's OHT bin assignment and its position within that bin. During backpropagation, the aforementioned OHT assignment reduces the multi-round construction to a single oblivious sort that places each request in its predetermined location. The scheduler for the parallel Neuron Fetcher is also retained, as the bin-to-{LSH-bucket, thread} mappings are unchanged.

Obliviousness. This optimization preserves obliviousness. The recorded layout information is consumed through the same fixed-size oblivious sorting procedure, so it does not introduce data-dependent branches or direct data-dependent addressing. The oblivious sorting operation moves the same requests into each bin as the original construction, and the subsequent Neuron Fetcher traverses the same LSH buckets and OHT bins under the same scheduler.

The only difference is the direction of data movement; neurons *transfer into* the OHT during the feedforward pass and *move back out* during backpropagation.

5.2.3 Neuron Updater. With the Request-Array fully populated by Neuron Fetcher, the Neuron Updater performs feedforward and backpropagation on the fixed-size Request-Array. Because requests are arranged in an FCFS order within Request-Array, the Neuron Updater processes them by linear scan rather than by a data-dependent traversal. Specifically, for each layer input i in a batch and layer number ℓ, TENNOR preallocates two arrays to store each fetched neuron's network-level index (or NeuronID) and its activation value, namely ActiveNodes$[i][\ell]$ and ActiveVals$[i][\ell]$. Both arrays have a fixed capacity of $R_\ell = \text{lenSeq}_\ell \cdot \text{PADSIZE}$, avoiding any data-dependent resizing or compaction during training.

During feedforward, the Neuron Updater linearly scans ReqArray_ℓ. For each neuron in each request's Neuron-Buffer, Neuron Updater computes the neuron's activation from the previous layer's active neurons, applies the activation function, and records the NeuronID and activation into ActiveNodes$[i][\ell]$ and ActiveVals$[i][\ell]$. During backpropagation, the updater traverses the same order—i.e., it computes deltas for each neuron in each request's Neuron-Buffer—, then runs nested loops whose bounds are determined solely by public parameters P to propagate errors and accumulate per-edge weight updates. See our extended manuscript [74] for more details.

Obliviousness. The Neuron Updater is oblivious because (1) all loop bounds depend solely on the public parameters P and (2) it linearly scans and updates the copies of neurons within each individual Request-Array entry. We use oblivious primitives for comparisons and conditional updates as opposed to data-dependent branches or memory accesses. Both feedforward and backpropagation follow a fixed, data-independent access pattern.

5.2.4 Oblivious Neuron Merge and LSH Rebuild. Before write-back, multiple requests may hold duplicate copies of the same neuron with updates from different inputs in the batch. To group these duplicate copies obliviously, TENNOR first applies OblSort, an *oblivious sorting primitive* implemented with a fixed data-independent compare and exchange pattern (e.g., bitonic sort [8]). TENNOR resolves these duplicates by obliviously grouping requests by LSHbucketID and then linearly scanning and consolidating to a single neuron copy action within each group (retaining the redundant entries as fake writes to preserve security). The merged results are then used to rebuild the OHT, which initiates a write-back pass that traverses the same LSH buckets and OHT bins as the feedforward fetch, with only the direction of neuron transfer reversed.

TENNOR also periodically rebuilds the LSH table by recomputing hash signatures for all neurons and relocating them in the new LSH table. During initialization, TENNOR adds PADSIZE dummy neurons per LSH bucket. Periodic rebuilds then reuse this fixed dummy reservoir while recomputing LSH signatures. If the sum of dummies and real neurons exceeds the capacity, oblivious compaction sends neurons to an (LSH) *overflow region*. The same refresh routine is used during initialization, with fresh dummies added in. The overflow region may hold real neurons, not just dummies, when more than PADSIZE hash to the same LSH bucket. Refer to our extended manuscript [74] for the merge and refresh routines, as well as for the empirical choice of PADSIZE.

Overflow Handling and Security. The overflow region has a fixed size (known via public parameters) and is present regardless of whether it contains real neurons or only dummy entries.

During every LSH initialization or refresh, TENNOR obliviously sorts all the neurons and executes a linear scan to obliviously mark the overflow neurons[2]. As a next step, TENNOR fills up every LSH bucket with real/dummy neurons (or both) and moves a fixed number of overflowing neurons (known via public parameters) to the overflow region (see our extended manuscript [74] for the corresponding algorithms). During training, TENNOR accesses only the fixed-size main region (LSH table) and ignores the overflow content. The overflowed neurons become available again during the next scheduled refresh. Overall, the overflow phenomenon may affect which neurons are updated between refreshes, but the handling of moving to and out of the overflow region is done obliviously. Thus, security is unaffected by actions in the overflow region. Our extended manuscript [74] empirically evaluates the effect of PADSIZE on the training accuracy of the neural network.

5.3 Implementation

We implement TENNOR in approximately 5,000 LoC in C++. The entire system runs in user space and requires no modifications to the OS kernel, TEE runtime, or system libraries. The implementation covers the full oblivious training pipeline, including MP-WTA hashing, the `Request-Array`, the OHT-based Neuron Fetcher, the fixed-shape Neuron Updater, and the oblivious merge and LSH refresh routines. We reuse only the network initialization and data-loading routines from SLIDE's open-source codebase [20].

For low-level oblivious primitives, we follow the design principles of Snoopy [29], which provides data-oblivious scalar and byte operations for TEEs. We extend this approach with oblivious routines for TENNOR's specific composite objects, such as request entries and neuron buffers. These routines are implemented within TENNOR's own codebase rather than by modifying Snoopy's library. Refer to our extended manuscript [74] for additional details.

6 Security Guarantees

Recall that P denotes the public parameter set (§5.2; see also our extended manuscript [74]). We model TENNOR as a *two-party computation* between a User and an OS/Hypervisor adversary $\mathcal{H}$. The User holds input (P, D), where D is the private training data; the Hypervisor holds input P.

Real World. The parties execute protocol π_{TENNOR} in the TD. The User sends (P, D) to the TD, which runs TENNOR and returns the trained model M to the User. The Hypervisor is passive. Its view consists of the public input P together with the memory-access trace τ observed during execution. We write: $\mathsf{view}_{\mathsf{H}}^{\pi_{\mathsf{TENNOR}}}(P, D) = \{P, \tau\}$.

Ideal World. The parties interact only through the functionality $\mathcal{F}_{\mathsf{train}}\big[(P, D),\ P\big] = \big[M,\ \perp\big]$. On input $((P, D), P)$, the functionality $\mathcal{F}_{\mathsf{train}}$ returns the trained model M to the User and returns $\perp$ to the Hypervisor. We say that π_{TENNOR} securely realizes $\mathcal{F}_{\mathsf{train}}$ if there exists a PPT simulator $\mathcal{S}$ such that, for all P and all D consistent with P, the joint output distribution in the ideal world is computationally indistinguishable from that in the real world:

$$\big\{\mathbf{Ideal}_{\mathcal{F}_{\mathsf{train}},\mathcal{S}}(\kappa;\ (P, D),\ P)\big\} \ \approx\ \big\{\mathbf{Real}_{\pi_{\mathsf{TENNOR}},\mathcal{H}}(\kappa;\ (P, D),\ P)\big\}$$

[2]Notice, though, that no neurons are permanently evicted from the table; thus, they have the chance to be considered after a rebuild.

Table 1: Overview of Evaluation Datasets.

Dataset	Feature-dim.	# Labels	#Train	#Test
Wiki10-31K	101,938	30,938	14,146	6,616
Amz-Sub	135,909	92,814	42,438	14,146
Wiki-325K	1,617,899	325,056	1,778,351	587,084
Amz-670K	135,909	670,091	490,449	153,025

In our setting, the Hypervisor output is $\perp$ in both worlds, and the security argument focuses on the information leaked through the Hypervisor's view, namely τ. The proof shows that the real Hypervisor view can be simulated from the public parameters P alone—i.e., τ does not reveal information beyond the leakage explicitly included in P. Equivalently, TENNOR securely realizes $\mathcal{F}_{\mathsf{train}}$ if there exists a PPT simulator $\mathcal{S}$ that, given only P, produces a simulated Hypervisor view such that:

$$\{\mathcal{S}(P)\} \approx_c \{\mathsf{view}_{\mathsf{H}}^{\pi_{\mathsf{TENNOR}}}(P, D)\}.$$

We prove Theorem 1 in the *hybrid model* [14], where self-contained primitives (i.e., oblivious sort, comparison, and hash table; refer to our extended manuscript [74]) are abstracted as ideal functionalities. Specifically, we prove that TENNOR is secure given *access to these functionalities*; each is then replaced by its corresponding protocol, which implements the functionality and whose security follows from prior work. The proof sketch and complete proof appear in our extended manuscript [74].

Theorem 1. *[Informal] Assume that each underlying oblivious primitive securely realizes its corresponding functionality.* TENNOR *then securely realizes* $\mathcal{F}_{\mathsf{train}}$ *against the Hypervisor in the hybrid model [14].*

7 Evaluation

We evaluate TENNOR to: (1) experimentally demonstrate the benefits of our approach and quantify the overall speedup over a baseline SLIDE+ORAM, which runs SLIDE [20] "as is" on top of a generic ORAM compiler [85] (§7.1); (2) attribute performance gains to individual optimizations (O1–O3, §5.2.2) via progressive ablations and runtime breakdowns (§7.2); and (3) measure the accuracy of `MP-WTA` compared to standard LSH (§7.3). The neural networks we train have a wide layer with 30K to 325K neurons.

Testbed. All experiments were conducted on one of two servers. For the performance and ablation experiments (1)–(2), we use a Google Cloud Confidential C3 instance equipped with 2×22-core/44-thread CPUs (Intel Sapphire Rapids) and 352 GB of RAM, running Ubuntu 24.04 LTS as a confidential VM image with Intel TDX as the TEE, and supporting AVX, AVX2, and AVX-512 instructions. For the accuracy evaluation (3), we use a bare-metal x86-64 server (64-core AMD EPYC 7763 Zen 3) without a TEE, as accuracy evaluation does not require secure execution. TENNOR is implemented in C++ (see §5.3) and compiled with GCC v13.3, with OpenMP enabled for multi-threaded parallelism. In addition, we enable AVX2/FMA support when available. Our oblivious primitives use inline assembly that is preserved at any optimization level. We compile without optimization flags (`-O0` under GCC) as an additional safeguard against compiler-introduced data-dependent branches for all tests.

Datasets. We use datasets from the Extreme Multi-Label Classification (XC) repository [10], as established in prior work [20]. For large-scale evaluation, we use WikiLSHTC-325K (Wiki-325K), a Wikipedia-derived dataset with 325K label categories. We also include Wiki10-31K, our smallest dataset, to evaluate TENNOR under a more constrained label space. Finally, we construct Amazon-Sub, a curated subset of Amazon-670K, with label cardinality falling between that of Wiki10-31K and Wiki-325K, serving as a medium-scale dataset. Together, the three datasets enable the evaluation of TENNOR across small, medium, and large label regimes that require a wide layer for the classification. Key statistics are in Table 1.

7.1 TENNOR vs. SLIDE+ORAM

Baseline. We compare TENNOR against SLIDE+ORAM, which uses Path ORAM [85] to store and access all neurons and LSH buckets used by SLIDE. We adopt Path ORAM as our baseline because it is an extremely simple tree-based ORAM construction, and has been widely used in secure-processor and enclave systems both in academia [76, 77, 85, 86] and industry [23]. In this baseline, neurons and LSH buckets are stored in a uniform representation on the ORAM server. Whenever SLIDE needs to read or write a neuron or an LSH bucket, it issues a corresponding ORAM access request through the ORAM client.

As discussed in Section 2.2, the ORAM client's memory accesses to the stash may still be observable via side-channels, even when running inside a TEE. To avoid the leakage from the stash accesses, the SLIDE+ORAM baseline performs a full linear scan over the stash on every access, avoiding data-dependent access patterns—a standard technique proposed by Mishra et al. [60]. Additionally, the baseline pads the number of neuron accesses per input to a fixed upper bound (i.e., the LSH bucket size) to conceal the distribution of retrieved neurons. We emphasize that this baseline is *not* doubly-oblivious (and thus has the potential to be faster), yet it represents a state-of-the-practice implementation that does not consider *eliminating* access-pattern leakage. Specifically, while it provides obliviousness for ORAM-backed storage accesses, it does not make the SLIDE computation itself oblivious: after retrieving neurons or buckets via the ORAM client, SLIDE's internal control flow and memory accesses still depend on the input, leaking information about the training data.

The baseline uses the same parameters as TENNOR (i.e., neural network architecture, LSH hash function count, and batch size), except that the number of LSH tables is 50, whereas TENNOR uses the newly proposed MP-WTA with only 1 hash table. All components—the neural network, Path ORAM, and SLIDE—run inside TDX.

Parameters. For all three datasets, we adopt the fully connected neural network with one wide hidden layer, similarly to the work of Chen et al. [20, 28]. For the chosen learning task, the wide layer is the output layer (i.e., where LSH is applied), which contains 30K, 90K, and 325K neurons for the Wiki-10, Amazon-Sub, and Wiki-325 datasets, respectively. As for the hidden layer, we chose a width of 128 for Amazon-Sub and Wiki-325K, matching the configuration used in prior work [20, 28]. Since Wiki10-31K is an additional dataset not evaluated in that work, we use a slightly larger hidden layer (256) to obtain a reasonable model capacity on this smaller label space while keeping the overall architecture consistent.

Table 2: Per-Batch Runtime (avg. over 1,280 inputs, batch size = 32). Speedup (SU) is relative to SLIDE+ORAM on the same dataset. Epoch (Ep.) time is extrapolated from mean batch time and epoch batch count. The baseline uses a sequential Path ORAM that does not benefit from multiple vCPUs.

Dataset	Method	#Thd	Batch(s/h)	SU	Ep.(h)
Wiki10	SLIDE+ORAM	1	2.6 h	1.0×	1,149
	TENNOR	8	681.3 s	13.7×	83.6
		22	338.8 s	27.6×	41.6
		44	248.3 s	37×	30.5
Amz-Sub	SLIDE+ORAM	1	16 h	1.0×	21,216
	TENNOR	8	2,085.5 s	27×	768.2
		22	1,007.8 s	57.1×	371.2
		44	695.7 s	82.7×	256.5
Wiki325	SLIDE+ORAM	1	208 h	1.0 ×	1.15×10^7
	TENNOR	8	4,717.8 s	157.2×	72,828
		22	2,255.4 s	328.8×	34,816
		44	1,578.3 s	469.9×	24,364

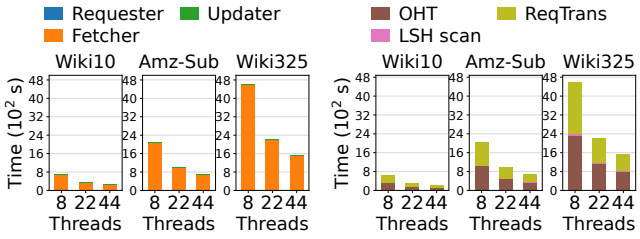
We fix the batch size to 32 across all datasets. This is dictated by our testbed constraints: Wiki-325K is our largest workload, and larger batch sizes exceed the available RAM when accounting for model state and associated data structures. Fixing the batch size also keeps comparisons consistent across datasets.

For LSH, we use WTA for the SLIDE+ORAM baseline and MP-WTA for TENNOR, applied to the output layer only, which is the primary computational bottleneck [20]. We select K via accuracy-driven tuning over a small set of candidates, yielding $K = 3$, 4, and 5 for Wiki10-31K, Amazon-Sub, and Wiki-325K, respectively, and use these values throughout all performance experiments.

To keep the Google Cloud infrastructure costs manageable, we benchmark a fixed window of 1,280 training inputs per dataset and configuration, reporting mean per-batch runtime over this window. We estimate end-to-end training time by scaling this mean by the total number of batches in each dataset's training schedule, determined by dataset size and epoch count. We evaluate TENNOR with 8, 22, and 44 threads, corresponding to real Google Cloud confidential computing configurations with varying core counts.

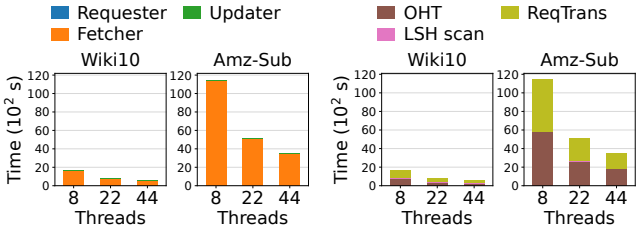
Performance Results. Table 2 compares the mean per-batch runtime of TENNOR against the baseline (SLIDE+ORAM) at batch size 32. Across all datasets, TENNOR achieves 1–3 orders-of-magnitude speedup, with gains increasing with additional CPU resources, reflecting effective parallelization of the dominant components. The baseline, by contrast, relies on a standard Path ORAM implementation with serialized accesses, so additional vCPUs offer limited benefit; we hence report a single baseline measurement per dataset.

On Wiki10-31K, the baseline takes 2.6 h, while TENNOR reduces runtime to 681 s, 338 s, and 248 s with 8, 22, and 44 vCPUs, corresponding to speedups of 13.7×, 27.6×, and 37×, respectively. On Amazon-Sub, TENNOR cuts the baseline runtime of 16 h down to 2,085 s, 1,007 s, and 695 s (8, 22, and 44 vCPUs), achieving speedups of 27×–82.7×. Although scaling is sublinear due to residual serial



(a) Total (b) Fetcher

Figure 2: Breakdown of Runtime (Total and Fetcher) across datasets and thread counts with batch size = 32.



(a) Total (b) Fetcher

Figure 3: Breakdown of Runtime (Total and Fetcher) across datasets and thread counts with batch size = 64.

Table 3: Runtime on Wiki10-31K and Amazon-Sub (batch size = 64). Wiki-325K is omitted due to RAM constraints.

Dataset	Method	#Thd	Batch(s/h)	SU	Ep.(h)
Wiki10	SLIDE+ORAM	1	5.2 h	1.0×	1,149
	TENNOR	8	1,655.8 s	11.3×	101.6
		22	795.9 s	23.5×	48.8
		44	565.4 s	33.1×	34.7
Amz-Sub	SLIDE+ORAM	1	32 h	1.0×	21,216
	TENNOR	8	11,496.9 s	10×	2,117.3
		22	5,185.5 s	22.2×	954.9
		44	3,514.4 s	32.7×	647.2

work and contention on shared, memory-intensive data structures, the absolute reduction in wall-clock time remains substantial across all core counts. On the larger Wiki-325K workload, the baseline requires 208 h, whereas TENNOR completes the same run in 4,717 s, 2,255 s, and 1,578 s with 8, 22, and 44 vCPUs, yielding speedups of 157.2×, 328.8×, and 469.9×.

These results demonstrate that TENNOR *makes training more feasible for substantially larger XC problems [10] where a Path ORAM-based approach becomes prohibitively slow.*

Following Table 2, we break down the mean per-batch runtime into major components to identify where time is spent. Figure 2 reports this component-level breakdown under the same measurement window. As shown in Figure 2a, the Neuron Fetcher dominates runtime, while the Neuron Requester, Neuron Updater, and remaining operations contribute negligible overhead, barely registering in the figure. Motivated by this, Figure 2b further decomposes the Neuron Fetcher into its internal stages. OHT construction and OHT-to-request conversion each account for nearly half of the Fetcher time, as both are dominated by computationally intensive oblivious sorting. The LSH scanning stage, by contrast, contributes only a small fraction of the Fetcher runtime. Overall, this breakdown pinpoints the post-optimization bottlenecks and motivates the progressive ablation analysis in Section 7.2.

Batch-size Sensitivity. We additionally evaluate a larger batch size of 64 on Wiki10-31K and Amazon-Sub (Table 3); Wiki-325K is omitted due to memory constraints on our evaluation testbed. While TENNOR continues to deliver substantial speedups (10×–33.1×), the gains are smaller than at batch size 32. As shown in Figure 3, this trend is consistent with the breakdown in Figure 2: the Neuron Fetcher is dominated by sort-heavy stages (OHT construction and OHT-to-request conversion) whose cost grows superlinearly with the number of items processed and becomes increasingly memory-bound at larger batch sizes. Hence, increasing the batch size amplifies the overhead of these stages and reduces parallel efficiency.

Training with a large effective batch size (B_{eff}) is nonetheless achievable using small micro-batches via gradient accumulation [34, 41, 66]: $B_{\text{eff}} = B_{\text{micro}} \times accum_steps$. Based on our evaluation, TENNOR can fix $B_{\text{micro}} = 32$ to maximize hardware efficiency and avoid the superlinear memory and computational overheads associated with larger micro-batches. Although we do not explicitly evaluate gradient accumulation in this work, it provides a straightforward mechanism to scale to the effective batch sizes required for convergence without increasing per-step system cost.

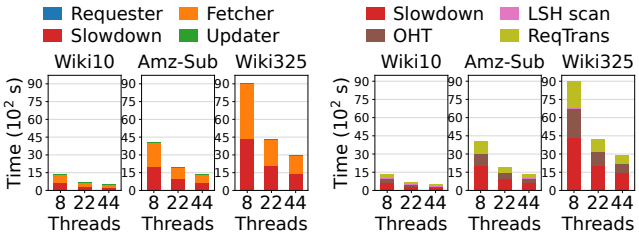
Memory Scaling. The OHT is the dominant batch-dependent component in peak memory, and it grows *quadratically* with the batch size B. Specifically, the OHT number of bins has a linear dependence on the total number of requests, i.e., $O(B \cdot \texttt{lenSeq})$, while the OHT per-bin capacity grows approximately linearly with $O(B \cdot \texttt{PADSIZE})$. Given that the OHT memory footprint is bins × per-bin capacity, for a fixed model configuration, $M_{\text{peak}} \approx M_{\text{fixed}} + O(B^2 \cdot \texttt{lenSeq} \cdot \texttt{PADSIZE})$. Our extended manuscript [74] reports the measured peak memory and provides a detailed analysis.

7.2 Optimization Ablations

We evaluate the performance impact of each optimization described in Section 5.2.2 using an ablation study. Unless stated otherwise, *All-Opt* enables all optimizations, and each ablated variant disables exactly one optimization while keeping the rest enabled.

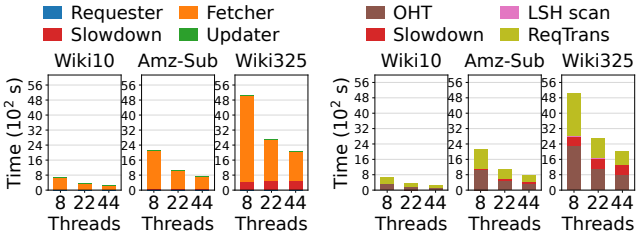
Impact of O1 (Deferred Dummy Payload). *NO-Opt-Payload-Deferral* disables optimization O1 (i.e., deferred dummy payload). Figure 4 shows the runtime breakdown, with the red segment representing the slowdown relative to *All-Opt*. Materializing dummy neuron payloads eagerly during OHT construction doubles the runtime, with the overhead concentrated in the OHT building stage.

Impact of O2 (Oblivious Parallelization). *NO-Opt-ParallelScan* disables optimization O2 (i.e., oblivious parallelization), reverting the Neuron Fetcher to a sequential LSH linear scan. Figure 5 shows the runtime breakdown, with the red segment indicating the overhead introduced by disabling this optimization.



(a) Total (b) Fetcher

Figure 4: Breakdown of Runtime (Total and Fetcher) without O1 (Deferred Dummy Payload) across datasets and thread counts with batch size = 32. (*NO-Opt-Payload-Deferral*)



(a) Total (b) Fetcher

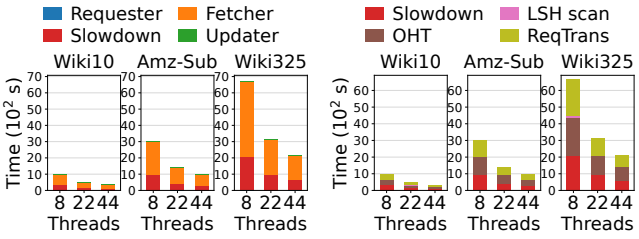
Figure 5: Breakdown of Runtime (Total and Fetcher) without O2 (Oblivious Parallelization) across datasets and thread counts with batch size = 32. (*NO-Opt-ParallelScan*)

As the dataset size increases, both the LSH tables and intermediate OHT grow, amplifying the cost of sequential scanning. Furthermore, as more threads are provisioned, the sequential scan becomes the dominant fraction of end-to-end runtime, since the remaining components parallelize well while the scan itself does not.

Impact of O3 (OHT Layout Reuse). *NO-Opt-OHTReuse* disables optimization O3 (i.e., OHT layout reuse), rebuilding the backpropagation OHT from scratch using the standard construction procedure. Figure 6 shows the runtime breakdown, with the red segment indicating the overhead relative to *All-Opt*. Disabling layout reuse incurs additional oblivious sorting and placement overhead, nearly doubling the OHT build time.

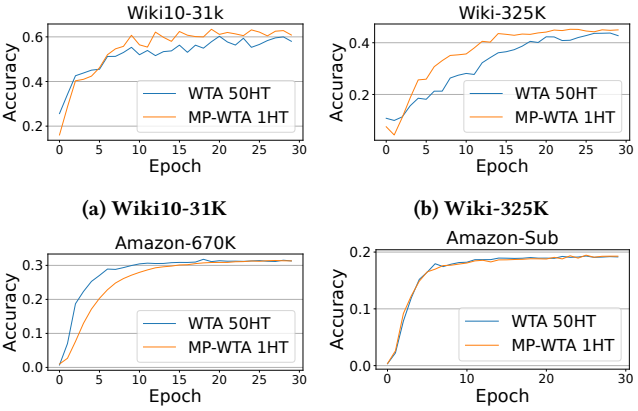
7.3 Accuracy of MP-WTA

We evaluate predictive accuracy at the algorithm level, comparing MP-WTA against standard LSH under identical training hyperparameters. Accordingly, we report accuracy using the baseline SLIDE implementation and its MP-WTA variant without TENNOR's secure-execution mechanisms (TEE/oblivious primitives), as our system transformations are semantics-preserving and affect only memory-access patterns. Thus, any accuracy differences observed here are attributable solely to the hashing scheme. Models range from 10M to over 100M parameters, with computation dominated by the final classification layer, isolating the effect of replacing standard LSH with MP-WTA on predictive performance across XC tasks.



(a) Total (b) Fetcher

Figure 6: Breakdown of Runtime (Total and Fetcher) without O3 (OHT Layout Reuse) across datasets and thread counts with batch size = 32. (*NO-Opt-OHTReuse*)



(a) Wiki10-31K (b) Wiki-325K

(c) Amazon-670K (d) Amazon-Sub

Figure 7: Classification accuracy of WTA (L=50) vs. MP-WTA (r=3, N=3, L=1) across all four datasets (window size M=8).

Setup. We isolate the effect of hashing by training two otherwise-identical models per dataset, sharing the same architecture, optimizer, and training schedule, and differing only in the final-layer hash function (WTA vs. MP-WTA). Unless stated otherwise, we use the same model configuration and hashing settings as in the performance evaluation (§7.1), with batch size as the only exception. Hashing is applied *only* to the final classification layer.

Hashing. We adopt the dataset-specific K from prior SLIDE work for Amazon-670K and Wiki-325K [28], and use the same values in our performance evaluation. For Wiki10-31K and Amazon-Sub, we select K via a lightweight sweep and fix it for both WTA and MP-WTA. Concretely, $K = 3, 4, 5, 6$ for Wiki10-31K, Amazon-Sub, Wiki-325K, and Amazon-670K, respectively. We cap WTA at $L = 50$ tables and set MP-WTA to a single table ($L = 1$), reflecting our compute/memory budget and testing whether MP-WTA retains accuracy with far fewer tables. Unless stated otherwise, MP-WTA uses multi-probe parameters $r = 3$, $N = 3$.

Hyperparameters. Each dataset uses a fully connected network with a single hidden layer, trained with Adam (learning rate 10^{-4}) [46]. Batch size is 128 for Amazon-670K, Wiki-325K, and Amazon-Sub, and 32 for Wiki10-31K.

Results. We report Precision at 1 (P@1)—i.e., the fraction of test samples whose top-predicted label matches a ground-truth label, following prior work [20, 28]. The results are averaged over 5 independent runs. As shown in Figure 7, across all four datasets, the MP–WTA ($r = 3, N = 3$) (1HT) curve closely tracks the WTA (50HT) baseline over training epochs. The gap narrows quickly in early epochs and remains small thereafter; in several cases the two curves become nearly indistinguishable by around ten epochs. With the window size held fixed, multi-probe substitutes for table multiplicity, yielding comparable accuracy while requiring roughly 50× fewer hash tables and proportionally less hash-table memory.

We also evaluate the time vs. accuracy results using the same runs as Figure 7, with epochs converted to wall-clock time. The corresponding results are reported in our extended manuscript [74]. *Compared with WTA, MP-WTA not only reduces storage requirements but also accelerates training time by using a single LSH table.*

8 Related Work

TEEs for ML. A substantial body of work leverages TEEs to protect models and inputs during ML training and deployment, typically via enclave-based ML services, model offloading, obfuscation, or partitioning across TEE/GPU configurations [7, 42, 68, 70, 87]. TEESlice [54] further demonstrates that when adversaries have access to public pre-trained models, such inference-time protections may fail. However, these methods primarily target model/input protection or *inference-time threats*, and do not address end-to-end *training-time leakage* through data-dependent access patterns.

Obliviousness for Learning. Prior studies have shown that enclave isolation alone is insufficient when attackers can observe memory access patterns. In response, researchers introduced algorithm-specific oblivious components for select learning tasks [71], including federated aggregation with sparsification [45]. Sparse and embedding-based models are especially vulnerable because their access patterns can reveal private features [38]. TENNOR treats *training-time access-pattern leakage* as a core threat and co-designs a dynamic sparse-training pipeline with integrated oblivious primitives, rather than safeguarding a single algorithm or aggregation step.

Oblivious Primitives and Operators. ORAM, O_2RAM, and oblivious data structures provide foundational access-hiding primitives [5, 60, 85, 86, 100]. LAORAM [75], in contrast, optimizes access locality in embedding-table training by grouping anticipated future accesses. Recent oblivious sorting, shuffling, and TEE-based parallel relational operators further improve the scalability of sorting- and operator-heavy routines [3, 36, 58, 69]. These works complement TENNOR at the primitive/operator level, while TENNOR presents a *system-level solution* for dynamic, sparse training.

Cryptographic PPML (MPC/HE). Approaches based on secure multiparty computation (MPC) and homomorphic encryption (HE) provide strong privacy guarantees without relying on trusted hardware, but tend to incur high computational costs, especially at scale [44, 64, 65, 90, 91]. TENNOR instead leverages TEEs for isolation and incorporates data-oblivious primitives, occupying a more practical point in the design space that balances privacy and performance for real-world training tasks.

9 Conclusion

We presented TENNOR, a CPU-friendly training system that enables the training of wide (and potentially sparse) neural networks within TEEs, while eliminating value-dependent NN access-pattern leakage across the whole memory hierarchy, including both enclave and external memory, beyond the publicly declared input support. TENNOR achieves this by co-designing the training pipeline with doubly-oblivious building blocks. It combines ORAM-backed storage with oblivious algorithms and data structures that make the memory behavior of each training iteration independent of the underlying data and model state. A key lesson from TENNOR is that practicality hinges on system-level co-optimization. Naively wrapping training with a baseline ORAM incurs prohibitive overhead, whereas exploiting the structure of training can substantially reduce redundant work and bring overhead to a more manageable level. Our evaluation demonstrates that TENNOR significantly outperforms standard Path ORAM-based baselines. TENNOR provides stable performance across datasets and thread configurations, while preserving the intended leakage profile. Looking forward, TENNOR is well-positioned to benefit from advances in TEE-friendly oblivious sorting and next-generation doubly-oblivious primitives, which can further accelerate the components that dominate today's costs.

Acknowledgments

We thank our anonymous shepherd and reviewers for their feedback. This work was supported by NSF awards #2238467, #2154732, and #2439951, and by a Commonwealth Cyber Initiative (CCI) grant from the Industry-Faculty Collaboration Acceleration Program (IF-CAP). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, or CCI.

References

[1] AMD. 2020. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. White Paper.
[2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing Systems*. 1225–1233.
[3] Gilad Asharov, T-H. Hubert Chan, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2020. Bucket Oblivious Sort: An Extremely Simple Oblivious Sort. In *Proc. of the Symposium on Simplicity in Algorithms*. 8–14.
[4] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: Optimal Oblivious RAM. In *Advances in Cryptology – EUROCRYPT*. 403–432.
[5] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. 2023. Futorama: A Concretely Efficient Hierarchical Oblivious RAM. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*. 3313–3327.
[6] Rohit Babbar and Bernhard Schölkopf. 2017. DiSMEC: Distributed Sparse Machines for Extreme Multi-label Classification. In *Proc. of the ACM International Conference on Web Search and Data Mining*. 721–729.
[7] Juyang Bai, Md Hafizul Islam Chowdhuryy, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. 2025. Phantom: Privacy-Preserving Deep Neural Network Model Obfuscation in Heterogeneous TEE and GPU System. In *Proc. of the USENIX Security Symposium*. 5565–5582.
[8] Kenneth E. Batcher. 1968. Sorting Networks and Their Applications. In *Proc. of the AFIPS Spring Joint Computer Conference*. 307–314.
[9] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. 2015. Conditional Computation in Neural Networks for Faster Models. arXiv:1511.06297
[10] K. Bhatia, K. Dahiya, H. Jain, P. Kar, A. Mittal, Y. Prabhu, and M. Varma. 2016. The Extreme Classification Repository: Multi-label Datasets and Code.
[11] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proc. of the Annual IEEE/ACM International Symposium on Microarchitecture*. 42–56.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proc. of the USENIX Workshop on Offensive Technologies*.

[13] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. 1998. Min-Wise Independent Permutations. In *Proc. of the Annual ACM Symposium on Theory of Computing*. 327–336.

[14] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology* 13, 1 (2000), 143–202.

[15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2015. Leakage-Abuse Attacks Against Searchable Encryption. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*. 668–679.

[16] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. *Proc. of the VLDB Endowment* 16, 13 (2023), 4324–4338.

[17] T-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2017. Oblivious Hashing Revisited, and Applications to Asymptotically Efficient ORAM and OPRAM. In *Advances in Cryptology – ASIACRYPT*. 660–690.

[18] Moses S. Charikar. 2002. Similarity Estimation Techniques from Rounding Algorithms. In *Proc. of the Annual ACM Symposium on Theory of Computing*. 380–388.

[19] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Ré. 2021. MONGOOSE: A Learnable LSH Framework for Efficient Neural Network Training. In *Proc. of the International Conference on Learning Representations*.

[20] Beidi Chen, Tharun Medini, James Farwell, Charlie Tai, Anshumali Shrivastava, et al. 2020. SLIDE: In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems. In *Proc. of the Annual Conference on Machine Learning and Systems*. 291–306.

[21] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjá Vu. In *Proc. of the ACM Asia Conference on Computer and Communications Security*. 7–18.

[22] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2024. Intel TDX Demystified: A Top-Down Approach. *Comput. Surveys* 56, 9 (2024), 1–33.

[23] Graeme Connell. 2022. Technology Deep Dive: Building a Faster ORAM Layer for Enclaves.

[24] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086. https://eprint.iacr.org/2016/086

[25] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proc. of the USENIX Security Symposium*. 857–874.

[26] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *Proc. of the ACM Conference on Recommender Systems*. 191–198.

[27] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*. 727–743.

[28] Shabnam Daghaghi, Tharun Medini, Nicholas Meisburger, Beidi Chen, Mengnan Zhao, and Anshumali Shrivastava. 2021. A Tale of Two Efficient and Informative Negative Sampling Distributions. In *Proc. of the International Conference on Machine Learning*. 2319–2329.

[29] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *Proc. of the ACM Symposium on Operating Systems Principles*. 655–671.

[30] Saba Eskandarian and Matei Zaharia. 2017. ObliDB: Oblivious Query Processing for Secure Databases. arXiv:1710.00458

[31] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[32] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *Proc. of the International Conference on Learning Representations*.

[33] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473.

[34] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677

[35] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2019. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *Proc. of the IEEE Symposium on Security and Privacy*. 496–512.

[36] Tianyao Gu, Yilei Wang, Afonso Tinoco, Bingnan Chen, Ke Yi, and Elaine Shi. 2025. Flexway O-Sort: Enclave-Friendly and Optimal Oblivious Sorting. In *Proc. of the USENIX Security Symposium*. 7563–7582.

[37] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *Advances in Neural Information Processing Systems*. 1135–1143.

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

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[40] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in Deep Learning: Pruning and Growth for Efficient Inference and Training in Neural Networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.

[41] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism. In *Advances in Neural Information Processing Systems*. 103–112.

[42] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. 2018. Chiron: Privacy-Preserving Machine Learning as a Service. arXiv:1803.05961

[43] Piotr Indyk and Rajeev Motwani. 1998. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proc. of the Annual ACM Symposium on Theory of Computing*. 604–613.

[44] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *Proc. of the USENIX Security Symposium*. 1651–1669.

[45] Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. 2023. Olive: Oblivious Federated Learning on Trusted Execution Environment Against the Risk of Sparsification. *Proc. of the VLDB Endowment* 16, 10 (2023), 2404–2417.

[46] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proc. of the International Conference on Learning Representations*.

[47] Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. In *Proc. of the International Conference on Learning Representations*.

[48] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2023. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In *Proc. of the USENIX Security Symposium*. 7285–7302.

[49] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2019. Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage. In *Proc. of the IEEE Symposium on Security and Privacy*. 1033–1050.

[50] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2020. The State of the Uniform: Attacks on Encrypted Databases Beyond the Uniform Query Distribution. In *Proc. of the IEEE Symposium on Security and Privacy*. 1223–1240.

[51] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks. In *Proc. of the IEEE Symposium on Security and Privacy*. 1502–1519.

[52] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *Proc. of the USENIX Security Symposium*. 487–504.

[53] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-Grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proc. of the USENIX Security Symposium*. 557–574.

[54] Ding Li, Ziqi Zhang, Mengyu Yao, Yifeng Cai, Yao Guo, and Xiangqun Chen. 2025. TEESlice: Protecting Sensitive Neural Network Models in Trusted Execution Environments When Attackers Have Pre-Trained Models. *ACM Transactions on Software Engineering and Methodology* 34, 6 (2025), 1–49.

[55] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-Based Power Side-Channel Attacks on x86. In *Proc. of the IEEE Symposium on Security and Privacy*. 355–371.

[56] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search. In *Proc. of the International Conference on Very Large Data Bases*. 950–961.

[57] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *Proc. of the USENIX Security Symposium*. 1289–1306.

[58] Apostolos Mavrogiannakis, Xian Wang, Ioannis Demertzis, Dimitrios Papadopoulos, and Minos Garofalakis. 2025. OBLIVIATOR: Oblivious Parallel Joins and Other Operators in Shared Memory Environments. In *Proc. of the USENIX Security Symposium*. 8521–8540.

[59] Tharun Kumar Reddy Medini, Qixuan Huang, Yiqiu Wang, Vijai Mohan, and Anshumali Shrivastava. 2019. Extreme Classification in Log Memory Using Count-Min Sketch: A Case Study of Amazon Search with 50M Products. In *Advances in Neural Information Processing Systems*. 13244–13254.

[60] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *Proc. of the IEEE Symposium on Security and Privacy*. 279–296.
[61] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. 2021. A Survey of Deep Learning on CPUs: Opportunities and Co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems* 33, 10 (2021), 5095–5115.
[62] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Proc. of the International Conference on Cryptographic Hardware and Embedded Systems*. 69–90.
[63] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *Proc. of the USENIX Security Symposium*. 469–486.
[64] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*. 35–52.
[65] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *Proc. of the IEEE Symposium on Security and Privacy*. 19–38.
[66] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proc. of the International Conference on Machine Learning*. 7937–7947.
[67] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, et al. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091
[68] Lucien K. L. Ng, Sherman S. M. Chow, Anna P. Y. Woo, Donald P. H. Wong, and Yongjun Zhao. 2021. Goten: GPU-Outsourcing Trusted Execution of Neural Network Training. In *Proc. of the AAAI Conference on Artificial Intelligence*. 14876–14883.
[69] Nicholas Ngai, Ioannis Demertzis, Javad Ghareh Chamani, and Dimitrios Papadopoulos. 2024. Distributed & Scalable Oblivious Sorting and Shuffling. In *Proc. of the IEEE Symposium on Security and Privacy*. 4277–4295.
[70] Yue Niu, Ramy E. Ali, and Salman Avestimehr. 2021. 3LegRace: Privacy-Preserving DNN Training over TEEs and GPUs. arXiv:2110.01229
[71] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proc. of the USENIX Security Symposium*. 619–636.
[72] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. 2018. PanORAMa: Oblivious RAM with Logarithmic Overhead. In *Proc. of the IEEE Annual Symposium on Foundations of Computer Science*. 871–882.
[73] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *Comput. Surveys* 51, 6 (2019), 1–36.
[74] Zifan Qu, Vasileios P Kemerlis, Giuseppe Ateniese, and Evgenios M Kornaropoulos. 2026. TENNOR: Trustworthy Execution for Neural Networks through Obliviousness and Retrievals. arXiv:2605.07160
[75] Rachit Rajat, Yongqin Wang, and Murali Annavaram. 2023. LAORAM: A Look Ahead ORAM Architecture for Training Large Embedding Tables. In *Proc. of the Annual International Symposium on Computer Architecture*. 1–15.
[76] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. 2013. Design Space Exploration and Optimization of Path Oblivious RAM in Secure Processors. In *Proc. of the Annual International Symposium on Computer Architecture*. 571–582.
[77] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *Proc. of the Network and Distributed System Security Symposium*.
[78] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*. 2565–2579.
[79] Erik Schultheis and Rohit Babbar. 2023. Towards Memory-Efficient Training for Extremely Large Output Spaces: Learning with 670k Labels on a Single Commodity GPU. In *Proc. of the Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. 689–704.
[80] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *Proc. of the International Conference on Learning Representations*.
[81] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proc. of the Network and Distributed System Security Symposium*.
[82] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proc. of the ACM Asia Conference on Computer and Communications Security*. 317–328.
[83] Anshumali Shrivastava and Ping Li. 2014. Asymmetric LSH (ALSH) for Sublinear Time Maximum Inner Product Search (MIPS). In *Advances in Neural Information Processing Systems*. 2321–2329.
[84] Ryan Spring and Anshumali Shrivastava. 2017. Scalable and Sustainable Deep Learning via Randomized Hashing. In *Proc. of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 445–454.
[85] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4 (2018), 1–26.
[86] Afonso Tinoco, Sixiang Gao, and Elaine Shi. 2023. EnigMap: External-Memory Oblivious Map for Secure Enclaves. In *Proc. of the USENIX Security Symposium*. 4033–4050.
[87] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *Proc. of the International Conference on Learning Representations*.
[88] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proc. of the USENIX Security Symposium*. 1041–1056.
[89] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*. 5998–6008.
[90] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. In *Proc. on Privacy Enhancing Technologies*, Vol. 2019. 26–49.
[91] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. In *Proc. on Privacy Enhancing Technologies*, Vol. 2021. 188–208.
[92] Xiao Wang, T-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*. 850–861.
[93] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*. 215–226.
[94] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks into Remote Timing Attacks on x86. In *Proc. of the USENIX Security Symposium*. 679–697.
[95] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. 2024. TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security*. 79–93.
[96] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of the IEEE Symposium on Security and Privacy*. 640–656.
[97] Jay Yagnik, Dennis Strelow, David A. Ross, and Ruei-sung Lin. 2011. The Power of Comparative Reasoning. In *Proc. of the IEEE International Conference on Computer Vision*. 2431–2438.
[98] Hsiang-Fu Yu, Kai Zhong, Jiong Zhang, Wei-Cheng Chang, and Inderjit S. Dhillon. 2022. PECOS: Prediction for Enormous and Correlated Output Spaces. *Journal of Machine Learning Research* 23, 98 (2022), 1–32.
[99] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. 2024. CacheWarp: Software-based Fault Injection using Selective State Reset. In *Proc. of the USENIX Security Symposium*. 1135–1151.
[100] Leqian Zheng, Zheng Zhang, Wentao Dong, Yao Zhang, Ye Wu, and Cong Wang. 2025. H2O2RAM: A High-Performance Hierarchical Doubly Oblivious RAM. In *Proc. of the USENIX Security Symposium*. 8501–8520.

A Open Science

The TENNOR artifact is publicly available.[3] We use the public XC benchmarks [10], and additionally provide Amazon-Sub with its generation script; the README documents the required toolchain, build process, and experiment commands. Reproducing the TEE experiments requires an Intel TDX platform with sufficient memory, while the non-TEE baseline runs on commodity hardware.

B Ethical Considerations

TENNOR is a defensive system evaluated using only public benchmarks and authorized infrastructure; it collects no user data and involves no human-subject experiments. In addition, TENNOR protects input-dependent memory-access patterns only under our threat model (§3). Our performance analysis reveals the cost of oblivious training, not attack or evasion tools.

[3] https://github.com/dorafan/TENNOR