Orca: FSS-based Secure Training with GPUs

Neha Jawalkar*†
Indian Institute of Science
jawalkarp@iisc.ac.in

Kanav Gupta*
Microsoft Research
kanav0610@gmail.com

Arkaprava Basu
Indian Institute of Science
arkapravab@iisc.ac.in

Nishanth Chandran
Microsoft Research
nichandr@microsoft.com

Divya Gupta
Microsoft Research
divya.gupta@microsoft.com

Rahul Sharma
Microsoft Research
rahsha@microsoft.com

***Abstract*—Secure Two-party Computation (2PC) allows two parties to compute any function on their private inputs without revealing their inputs in the clear to each other. Since 2PC is known to have notoriously high overheads, one of the most popular computation models is that of 2PC with a trusted dealer, where a trusted dealer provides correlated randomness (independent of any input) to both parties during a preprocessing phase. Recent works construct efficient 2PC protocols in this model based on the cryptographic technique of function secret sharing (FSS).**

We build an end-to-end system ORCA to accelerate the computation of FSS-based 2PC protocols with GPUs. Next, we observe that the main performance bottleneck in such accelerated protocols is in storage (due to the large amount of correlated randomness), and we design new FSS-based 2PC cryptographic protocols for several key functionalities in ML which reduce storage by up to $5\times$. Compared to prior state-of-the-art on secure training accelerated with GPUs in the same computation model (PIRANHA, Usenix Security 2022), we show that ORCA has 4% higher accuracy, $123\times$ lesser communication, and is $19\times$ faster on CIFAR-10.

I. Introduction

Machine learning training has emerged as an extremely important and data hungry application. While the trained models get better with more data and diverse data, very often this data is highly sensitive, e.g., financial, healthcare or browsing data. Use of privacy-preserving technologies such as secure multiparty computation (MPC) [30], [66] provide secure training over this sensitive data [39], [46], [47], [58], [60], [62], [64]. Secure training using MPC allows multiple mutually distrusting parties to train a model of their joint data without revealing anything about their data to each other or to any other party beyond the final trained model. However, MPC-based secure training has high performance overheads. Through a series of recent works, the end-to-end time required to securely train models on the CIFAR-10 dataset has reduced from years [47], to months [62], to weeks [58], to a day with PIRANHA [64]. PIRANHA is the current state-of-the-art in accelerating secure training using GPUs. Our goal is to reduce these overheads to an hour.

PIRANHA works in the *pre-processing model* where the data-independent pre-processing material (or, correlated randomness) is provided by a trusted dealer in an offline phase, and parties use this material during the data-dependent online phase. Works in this model, including PIRANHA and us, focus on reducing the online complexity. To obtain efficient secure training, PIRANHA makes several MPC-friendly approximations to ML such as replacing maxpools with average pools [58], local truncations [47], [58] and approximating exponentiations with piecewise-linear functions. Moreover, following Falcon [62], they also trade-off security for efficiency, and reveal intermediate values during softmax computation which is a leakage disallowed by standard MPC security requirements. PIRANHA observes that its ad hoc approximations lead to a significant loss in model accuracy w.r.t. PyTorch training. This in turn implies that PIRANHA has to approximate large models, e.g., VGG16, which give 67% accuracy in PyTorch on CIFAR, to obtain a reasonable accuracy of 55% with secure training, i.e., the accuracy loss w.r.t. PyTorch training is as large as 12%.

A. Our Contributions

Unlike [47], [58], [60], [62], [64], our secure training remains faithful to quantized training algorithms from ML literature that are known to mirror floating-point PyTorch training [32]. We show that faithful secure training of small models (with thousands of parameters) produces more accurate models than PIRANHA training large models (with millions of parameters). PIRANHA approximations fail on small models - training our small models with these reduces the accuracy of the trained model to that of a random classifier. On CIFAR, our secure training beats PIRANHA on all metrics. We are more secure (as we don't reveal any intermediate value) and produce models with 4% higher accuracy in $19\times$ less time while incurring $123\times$ lower communication. In absolute terms, ORCA reaches a CIFAR accuracy of 59% in an hour and 70% within 3 hours (vs. PIRANHA's 55% in a day).

On the technical side, our starting point is recent advances in function secret sharing (FSS) based secure 2-party computation (2PC) protocols in the dealer model. A key feature of these protocols is that they reduce online communication while increasing compute and storage. The online phase in FSS requires a large number of AES evaluations and reads huge FSS keys that are generated during pre-processing. Thus, FSS

* Equal contribution.

† Work partially done while at Microsoft Research.

shifts the performance bottleneck from the external network to compute and memory that is scoped to a single machine. In this work, we significantly reduce the overheads of secure training by effectively accelerating FSS-based 2PC with GPUs. However, we face several challenges on both the systems and cryptographic front, which we discuss next.

1) System optimizations: To accelerate compute, we create efficient GPU implementations of FSS protocols (Section III) that are an order of magnitude faster than the CPU-based protocols. This result is surprising because prior work that attempted to accelerate FSS with GPUs got only marginal improvements over the CPU implementation [55]. The key to our accelerated implementation is a unique combination of system optimizations guided by the GPU micro-architecture (Section III) and new cryptographic techniques (Section V). We leverage several GPU micro-architectural features, such as using GPU's scratchpad memory for faster AES computation, optimizing data layout to improve GPU cache hit rates, and utilizing GPU's lockstep execution by groups of threads to optimally pack intermediate results of cryptographic computations in the memory (detailed in Section III). Interestingly, we discovered that once the computation is well-optimized on the GPU, the time to read GBs of FSS keys from the SSD to GPU memory becomes the bottleneck. To address this, we lean on new cryptographic protocols (discussed next) to reduce key size for commonly occurring nodes in training.

2) Cryptographic improvements: Faithful quantized training, e.g., Gupta et al. [32], requires efficient protocols for *stochastic truncations* of fixed-point values, maxpools, and floating-point softmax. Stochastic truncations and maxpools are expensive[1] and lead to large keys. We create novel protocols that reduce the key size by up to $5\times$ (Section V). Quantized training performs almost all operations in fixed-point but uses floating-point arithmetic in softmax computations to maintain accuracy [32]. In secure training literature, there are three ways of computing exponentiations occurring in softmax. First, use the very cheap piecewise-linear approximations [40], [64]. Second, use fixed-point approximations that are more expensive [38], [39]. Third, use precise computation over floating-point [51]. To stay faithful to the computations done by Gupta et al. [32], we choose the third option. To this end, we create novel FSS protocols to efficiently stitch protocols for fixed-point with protocols for floating-point (Section VI).

3) ORCA: We implement our techniques in ORCA, a push-button tool for secure training, that will be made publicly available. ORCA sports a library of GPU-accelerated FSS blocks that future research in this area can build upon (Section VII). We show that training small models with ORCA while staying faithful (see Figure 5) to quantized training [32], outperforms the approximate large model training of PIRANHA in accuracy (upto 4%), time $(4-19\times)$, and communication $(51-123\times)$ (Section VIII-A). In a more apples-to-apples comparison, ORCA outperforms PIRANHA on approximate training of large models by $2.3-4.9\times$ (Section VIII-B1). ORCA also outperforms prior works that don't use PIRANHA approximations by $7-441\times$ (Section VIII-B2). ORCA's protocols need smaller FSS keys (by $5\times$, Table VII) and ORCA's GPU-based protocols outperform their CPU counterparts by an order of magnitude (Table VI). Finally, ORCA achieves the first ever sub-second ImageNet-scale inference (13s for a batch of 16 images) by outperforming the state-of-the-art [31] by over $50\times$ (Table VI).

[1]High MPC cost of these functions motivated the approximations in [64].

II. PRELIMINARIES

A. Notation

Let λ denote the computational security parameter. $1\{b\}$ represents the indicator function that outputs 1 when the predicate b is true and 0 otherwise. Arrays are represented using boldface, and their elements are represented using the same symbol in a normal typeface with the index in subscript, starting from 0. For example $\boldsymbol{x} = \{x_0, x_1, \dots\}$.

a) Datatypes: For $N = 2^n$, $\mathbb{U}_N$ represents the set of n-bit unsigned integers. We denote the set of real numbers using the symbol $\mathbb{R}$. For $x \in \mathbb{U}_N$, $\mathsf{uint}_n(x)$ and $\mathsf{int}_n(x)$ represent the corresponding unsigned and signed number (in 2's-complement representation) in $\mathbb{Z}$ respectively.

Fixed-point representation. Fixed-point representation, parameterized by a bitwidth n and a scale f, encodes a real number $r \in \mathbb{R}$ into $x \in \mathbb{U}_N$ such that $x = \lfloor r \cdot 2^f \rfloor \mod N$. For an unsigned (resp., signed) fixed-point number $x \in \mathbb{U}_N$ with scale f, $[\![x]\!]^+_{n,f}$ (resp., $[\![x]\!]_{n,f}$) denotes it's underlying real value $\frac{\mathsf{uint}_n(x)}{2^f}$ (resp. $\frac{\mathsf{int}_n(x)}{2^f}$).

b) Operators: Arithmetic operations in $\mathbb{U}_N$, like addition and multiplication, are followed by a $\mod N$ and we omit this whenever it is clear from the context. For $x \in \mathbb{U}_M$ and $n > m$, we use the notation $\mathsf{ZeroExt}_{m,n}(x)$ (resp., $\mathsf{SignExt}_{m,n}(x)$) to represent a number $y \in \mathbb{U}_N$ such that $\mathsf{uint}_m(x) = \mathsf{uint}_n(y)$ (resp., $\mathsf{int}_m(x) = \mathsf{int}_n(y)$). We use $x \gg_A f$ (resp., $x \gg_L f$) to represent arithmetic (resp., logical) right shift of x by f such that the input and the output have the same bitwidth. For an n-bit number x and $f < n$, the operation truncate-reduce, denoted by $\mathsf{TR}(x, f)$, is defined as dropping the lower f bits of input and returning the output as a $(n-f)$-bit number. For an array $\boldsymbol{a}$ and $i < |\boldsymbol{a}|$, we use the notations $\boldsymbol{a} \ggg i$ and $\boldsymbol{a} \lll i$ to represent the array rotated to the right and left, respectively, by i steps.

c) Secret sharing: For $x \in \mathbb{U}_N$, we define (additive) secret sharing of x as the process of sampling two random numbers $x_0, x_1 \in \mathbb{U}_N$, such that $x = x_0 + x_1 \mod N$ and denote it by $\mathsf{share}(x)$. For array variables and tuples, this operation is applied element-wise. When the secret shares are held by two parties, e.g., P_0 holds x_0 and P_1 holds x_1, we denote the operation of exchanging the shares and adding them up by $x = \mathsf{reconstruct}(x_b)$, for $b \in \{0, 1\}$.

B. Function Secret Sharing

A Function Secret Sharing (FSS) Scheme [19], [20] is a pair of algorithms $(\mathsf{Gen}, \mathsf{Eval})$. Gen splits a function $g : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$ into two functions (g_0, g_1) and Eval takes as input the party identifier $b \in \{0, 1\}$, function share g_b and evaluates g_b

on input $x \in \mathbb{G}^{\mathsf{in}}$. The correctness property of the FSS scheme requires $g_0(x) + g_1(x) = g(x)$. The security property of the FSS scheme requires that each function g_b hides g.

Definition 1 (FSS: Syntax [19], [20]). *A (2-party) FSS scheme is a pair of algorithms* $(\mathsf{Gen}, \mathsf{Eval})$ *such that:*

- $\mathsf{Gen}(1^\lambda, \hat{g})$ *is a PPT key generation algorithm that given* 1^λ *and* $\hat{g} \in \{0,1\}^*$ *(description of a function* g*) outputs a pair of keys* (k_0, k_1)*. We assume that* $\hat{g}$ *explicitly contains descriptions of input and output groups* $\mathbb{G}^{\mathsf{in}}, \mathbb{G}^{\mathsf{out}}$.
- $\mathsf{Eval}(b, k_b, x)$ *is a polynomial-time evaluation algorithm that given* $b \in \{0,1\}$ *(party index),* k_b *(key defining* $g_b : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$*) and* $x \in \mathbb{G}^{\mathsf{in}}$ *(input for* g_b*) outputs* $y_b \in \mathbb{G}^{\mathsf{out}}$ *(the value of* $g_b(x)$*).*

The keys (k_0, k_1) output by Gen are called FSS keys. The size of k_0 or k_1 is the *key size* and corresponds to the preprocessing material required to be stored by a single evaluator. We formally define the correctness and security properties of an FSS scheme in Appendix B.

C. Distributed Comparison Function (DCF)

DCFs were introduced by Boyle et al. [20] and provides an FSS scheme for special interval functions.

Definition 2 (DCF [18], [20]). *A special interval function* $f^{<}_{\alpha,\beta} : \mathbb{U}_N \to \mathbb{G}^{\mathsf{out}}$*, also called comparison function, takes as an input* $x \in \mathbb{U}_N$ *and outputs* β *if* $x < \alpha$ *and* 0 *otherwise. The corresponding FSS scheme for this function* $(\mathsf{Gen}^{<}, \mathsf{Eval}^{<})$ *is called Distributed Comparison Function.*

All our protocols use DCFs for the case when $\mathbb{G}^{\mathsf{out}} = \mathbb{U}_L$ for some $L = 2^\ell, \ell \leqslant \lambda$. Below, we summarize the cost of such a DCF using the optimized construction from [18].

Theorem 1 (Cost of DCF [18]). *Given a PRG* $G : \{0,1\}^\lambda \to \{0,1\}^{4\lambda+2}$*, there exists a DCF for* $f^{<}_{\alpha,\beta} : \mathbb{U}_N \to \mathbb{U}_L$ *for* $N = 2^n, L = 2^\ell$ *with key size* $n(\lambda + \ell + 2) + \lambda + \ell$*. The number of PRG invocations in* $\mathsf{Gen}^{<}_n$ *is* $2n$ *and in* $\mathsf{Eval}^{<}_n$ *is* n*.*

In the rest of the paper, we use the notations $\mathsf{DCF}_{n,\ell}$ and $\mathsf{keysize}(\mathsf{DCF}_{n,\ell})$ to represent this scheme and its key size, respectively. Similar to prior works [18], [31], we set $\lambda = 126$ and realize the required PRG using 4 calls to AES with 128-bit output in CTR mode. However, as observed in [31], during evaluation it suffices to make only 2 AES calls because two out of the four blocks generated are discarded in the evaluation algorithm of [18] and CTR mode allows us to generate the required blocks without generating the other two.

D. Secure 2PC with preprocessing using FSS

Threat model. We consider 2-party secure computation (2PC) in the trusted dealer model. That is, there exists a trusted dealer that provides correlated randomness to the two parties in a preprocessing phase (before inputs to the computation are available). We prove the security of our protocols in the standard simulation paradigm [22], [30], [44] against a semi-honest static probabilistic polynomial time (PPT) adversary that corrupts one of the two parties. For completeness, we describe the detailed threat model in Appendix C.

Our protocols trivially extend to the "client-server" model, where $m \geqslant 2$ clients secret share their inputs with the servers P_0 and P_1, thus delegating their computation to these two parties. Security can be analogously defined in this case and we can obtain semi-honest security against up to $m - 1$ clients colluding with one of the two servers.

2PC from FSS. [18], [21] proposed a semi-honest static secure 2PC in the trusted dealer model using FSS. Consider the scenario when the two evaluators want to evaluate the computation circuit with gates $\{g_i\}_i$ and wires $\{w_i\}_i$. We describe the protocol from [18], [21] to evaluate this circuit securely in two phases - preprocessing and online.

1) Preprocessing Phase: For each wire w_i in the computation circuit, the dealer randomly samples a mask r_i. For each gate g, with input wire w_i and output wire w_j, the dealer generates FSS keys (k_0^g, k_1^g) for the *offset function*, $g^{[r_i,r_j]}(x) = g(x - r_i) + r_j$, and sends one key to each party. For input and output wires w_i owned by party b, it sends the corresponding mask r_i to party b.

2) Online Phase: For each input wire w_i with value x_i owned by party b, the party b calculates masked wire value $\hat{x}_i = x_i + r_i$ and sends it to the other party. Now, starting from the input gates, the two parties process gates in topological order to receive masked output wire values. To process a gate g, with input wire w_i, output wire w_j, and masked input wire value $\hat{x}_i = x_i + r_i$, party b uses Eval with k_b^g and $\hat{x}_i$ to obtain a share of the masked output wire value $\hat{x}_j = g^{[r_i,r_j]}(\hat{x}_i) = g(\hat{x}_i - r_i) + r_j = g(x_i) + r_j = x_j + r_j$ which they reconstruct using a single round of communication to obtain $\hat{x}_j$. For output wires, they subtract the corresponding mask received from the dealer to obtain clear output values.

E. GPU accelerated computing

While the graphics processing unit (GPU) was originally designed for accelerating graphics, they have emerged as a key platform to accelerate parallel computation, including DNN training, data analytics, graph processing, etc. Even a mid-range NVIDIA A6000 GPU can execute over five thousand threads to execute simultaneously. Further, more than a hundred thousand threads remain ready to run in a GPU for quickly substituting threads whose execution stalls. GPU programming languages such as CUDA [2] require programmers to arrange threads in a hierarchy to keep GPU's massive parallelism tractable. A group of 32 GPU threads called warp typically executes in lockstep and is the smallest schedulable unit of work. Up to 32 warps make up a threadblock. A GPU kernel is often launched with hundreds of threadblocks, creating a grid of hundreds of thousands of threads.

A GPU's architecture reflects its programming hierarchy. Threads of a warp execute in lockstep on a Single Instruction Multiple Data (SIMD) unit. Several such SIMD units are placed on a Streaming Multiprocessor (SM). A GPU will have tens of such SMs; e.g., A6000 has 84 SMs. All threads of a threadblock are scheduled onto the same SM, while threads from different threadblocks can run on different SMs.

GPU's memory subsystem reflects a similar hierarchy. A GPU would typically sport onboard memory with more than TB/sec bandwidth but limited to a few tens of GBs in capacity. Contents from this memory are cached in two levels of hardware caches. A GPU has several specialized software-managed hardware caches too. For example, each SM has a scratchpad (a.k.a, shared memory) for easy sharing of data amongst the threads of the *same* threadblock. This is possible since all threads of a threadblock are guaranteed to execute on the same SM. Similarly, a constant cache in an SM is purpose-built for storing frequently used read-only data. Each thread in CUDA also has tens of fast registers. Since threads of a warp always execute as a group, CUDA enables collective operations to exchange and/or reduce data across threads of a warp using the registers.

Finally, we note that a GPU is always accompanied by a CPU. It connects to its host CPU over a PCIe interconnect [10]. A GPU-accelerated program starts running on the CPU. Portions of the program running on the CPU allocates/de-allocates memory on the GPU and also transfer data to and from the GPU memory over the PCIe bus. CPU is also responsible for launching GPU "kernels" (GPU-accelerated function written in CUDA/OpenCL) with a desired number of threads in the grid to compute on the GPU.

III. Accelerating FSS on a GPU

Accelerating FSS-based secure computation on GPU is an essential goal of Orca. Toward this, we make progress in two key aspects. ① We demonstrate how a GPU's architecture must be leveraged for reasonable speedups in FSS-based computation. ② We discover that the time to read FSS keys from the storage can eclipse the benefits of GPU acceleration. We propose a combination of new cryptographic techniques and systems improvisations to limit key read time. Next, we detail these two aspects in Orca's design.

A. Accelerating FSS-based compute on GPU

The prior work that attempted to accelerate FSS-based protocol on GPU [55] failed to observe any significant speedup over CPU implementation in the absence of a comprehensive strategy to leverage idiosyncrasies of a GPU architecture. In Orca, we employ three key techniques to harness GPU's computing power as follows.

❶ Faster AES computation (AES) A key primitive in FSS-based computation is the Distributed Comparison Function or DCF (Section IV). We empirically find that computing DCF can account for the majority of overall computation time. For example, in the forward pass of CNN3, computing DCFs accounts for about 93% of the overall compute time. Evaluating a DCF requires $2n$ invocations of AES (Section II-C).

In the rest of this subsection, we consider the task of performing 10 million DCF evaluations as a microbenchmark to quantify the speed-up potentials of different optimizations we will discuss in this subsection. This choice is driven by the fact that several models perform many millions of DCFs per layer. Table I empirically captures the computation time of the microbenchmark, starting with the baseline discussed next.

To accelerate AES on a GPU, we start by using PyTorch's `csprng` extension [11] following prior work [55], [58]. AES requires repeatedly looking up precomputed lookup tables [59]. Upon analyzing the performance of PyTorch's `csprng` using NVIDIA's Nsight tool [8], we notice that it often stalls while accessing the lookup table. It keeps lookup tables on the constant cache within each SM of the GPU. While accesses to constant cache are fast, they are suitable only if the GPU threads access the *same address* at any given cycle [3]. Otherwise, the accesses are serialized, stalling computation. Unfortunately, different threads access different indexes (thus, different addresses) in the lookup table.

To reduce such stalls, we replicate the lookup table once for each warp in an SM (here, 32) following the strategy laid out in a prior work [59]. Further, the replicated tables are placed onto the scratchpad (shared memory) of each SM in the GPU. This is because the scratchpad is banked, unlike the constant cache. Data in different banks can be accessed simultaneously without stalling. Thus, replicas of the lookup table are placed in different banks of the scratchpad, alleviating stalls due to accesses to the AES's lookup table.

The first entry in Table I shows that the AES implementation of PyTorch's `csprng` extension [11] requires 3305 ms. It reduces to 840 ms when we use the optimized AES implementation ("AES" in the Table) giving a $3.9\times$ speedup.

❷ Optimized data layout for cache locality (LAYOUT) Computing a DCF requires the evaluator to perform n *chained* PRG ($2n$ AES) [18], [31]. However, the evaluator slightly modifies the output of the i^{th} PRG invocation before feeding it to the $(i+1)^{th}$ PRG invocation with a *correction word* (CW_i) [18]. Consequently, there are n correction words for each DCF key. We notice that the layout of these correction words in the memory impacts performance.

In a parallelized CPU implementation, each thread would compute a DCF while running independently on a CPU core. Thus, laying out n correction words for a DCF computation contiguous in the memory for better cache locality.

In a GPU implementation, however, a group of 32 threads in a warp execute in lockstep as each thread computes a DCF. In a lockstep execution, threads in a warp can proceed to compute its k^{th} PRG only when all threads in the warp have finished computing $(k-1)^{\text{th}}$ PRG. Thus, unlike the CPU implementation, keeping all the n correction words for a given DCF contiguously leads to poor cache hit rates. Instead, for a GPU implementation, one must consider cache locality for all threads in a warp in aggregate. Therefore, we place the correction words for a given round of PRG across all threads in a warp contiguously. In other words, the correction words for a given (say, k^{th}) round of PRG of *all* threads in the warp (DCF computation) are laid contiguously in the memory. The correction words for the next round of PRG ($k{+}1^{\text{th}}$) are placed thereafter in the memory in a similar fashion.

We found that the optimized layout of correction words

	Naive	AES	AES+LAYOUT	AES+LAYOUT+MEM
Time (ms)	3305	840	716	523
Speedup		$3.9\times$	$4.6\times$	$6.3\times$

Table I: Speedup of 10M DCFs with our optimizations

improves the L1 cache hit rate from 20% to 49%. As shown in Table I, this optimization (LAYOUT) further reduces the time to compute ten million DCFs from 840 ms to 716 ms.

③ Optimizing memory footprint (MEM): FSS-based protocols lower communication overheads but demand larger key sizes. The files containing keys would typically reside on the storage (e.g., SSD) and must be read into the memory during online computation. Reading large keys from the SSD can be prohibitively slow (next subsection). We address this challenge with novel cryptographic protocols for commonly occurring nodes in training detailed in Sections IV, V and VI. A key technique is using DCFs with smaller payloads, e.g., 1-bit instead of 64-bit, and performing comparisons on shorter inputs, e.g., 40-bit inputs instead of 64-bit inputs wherever possible. Overall we achieve significant reductions in key size depending on the function, e.g., $5.8\times$ reduction for truncation followed by ReLU. Since memory is limited on a GPU, the smaller payload for DCF key can help reduce the memory footprint of GPU computation. It can also limit data movement between the CPU and GPU over the slow PCIe interconnect.

However, harnessing the full benefits of a smaller DCF payload is not straightforward. A simple implementation that would keep 1-bit output to a standard data type such as a byte leads to $8\times$ memory bloat. Instead, in ORCA, 32 threads of a warp write their 1-bit DCF output into a 32-bits integer in a lockstep fashion. This avoids memory bloat but requires threads in a warp to write to a single integer without needing locks. Locks are prohibitively slow in GPUs [63]. ORCA leverages CUDA's warp-synchronization primitives to ensure that each thread can write its output without interfering with writes from other threads in the warp. Specifically, it uses CUDA's __ballot_sync() and __shfl_down_sync() intrinsic methods for warp-level synchronized data exchange without locks [13].

Table I shows further speedup thanks to this optimization (MEM). The time to perform ten million DCFs drops to 523 ms (27% speedup), capping a $6.3\times$ improvement over the naive implementation of DCFs on the GPU.

B. Reducing time to read FSS keys

A key advantage of FSS-based secure computing protocol is that it limits communication between the parties. However, it necessitates reading large amounts of pre-generated keys while performing online computation. We discovered that once the computation is accelerated on the GPU through the above-mentioned strategies, the time to read the FSS keys from the storage (here, SSD) to GPU's memory becomes the bottleneck. We adopt a three-prong approach to address this bottleneck.

① Bypassing OS page cache: By default, the OS caches file contents on the CPU's DRAM in the hope that a file's data will be accessed repeatedly over time. However, page cache can add overhead in the critical path of accessing file contents. Since an FSS key is used only once, there is no reuse. Thus, files containing FSS keys don't benefit from the page cache but pay the overheads. For example, one of the models that we train, P-VGG16 from prior work [64], needs 17GB of key per iteration. The time for reading this key reduces from 19.7s to 6.3s, thanks to bypassing the OS's page cache.

② Overlapping key read with computation: To further reduce the impact of the key read time, we overlap the computation of the i^{th} training iteration with the reading of the key for the $(i+1)^{th}$ iteration. This ensures that the *entire* key read time is not in the critical path.

③ New cryptographic technique to limit key size: Even after the above-mentioned optimizations, the time to read the keys from the storage can overshadow the computation time on the GPU for training larger networks. We observe that key read time becomes the bottleneck only when the computation is accelerated well on the GPU as in ORCA. This is not the case for a CPU-only implementation of FSS protocols or for the GPU-based implementation without the optimizations discussed in this subsection.

In ORCA, we construct new FSS protocols to reduce keysize in popular non-linearities like ReLU and maxpool that result in reduced online communication as well when combined with truncations that need to be performed for fixed-point training (Sections IV,V). Overall, as we show in Section VIII, over the entire network, we get up to $3.4\times$ reduction in key size.

IV. Protocols for Basic Building Blocks

Protocols in this section and in Sections V and VI will use the following protocol syntax.

Syntax. We use $\hat{(\cdot)}$ to denote masked values, e.g., $\hat{x}$. We describe our protocols (denoted by Π) for the setting when evaluators hold masked values of input x, denoted by $\hat{x}$, and the dealer holds the mask r^{in}. After the protocol, the evaluators hold the shares of the masked output, that is, $\hat{y} = y + \mathsf{r}^{\mathsf{out}}$ for output y. The corresponding protocols $\hat{\Pi}$ where evaluators end with masked output $\hat{y}$ in the clear can be easily constructed from Π by adding a round of $\mathsf{reconstruct}$. We denote the key size required by the protocol Π by $\mathsf{keysize}(\Pi)$.

Unlike the previous works on FSS-based 2PC [18], [21], [31] where the online phase for a gate was non-interactive, in this work, we construct more complex protocols, where the online phase is allowed to be multi-round. However, this does not pose any issue w.r.t. stitching together protocols as evaluators still end up with masked values or shares of the same, where the mask is known to the dealer.

Below, we first provide new FSS-based protocols for simple functions Select and Signed Extension that would be used as building blocks in the protocols described in later sections. Later, we summarize the result from prior work for multiple interval containment that will also be used as a building block.

A. Select

Select function, $\mathsf{select}_n : \{0,1\} \times \mathbb{U}_N \rightarrow \mathbb{U}_N$, takes as input a selector bit $s \in \{0,1\}$ and an n-bit payload $x \in$

```
Select $\Pi_n^{\mathsf{select}}$
$\mathsf{Gen}_n^{\mathsf{select}}((\mathsf{r}_1^{\mathsf{in}}, \mathsf{r}_2^{\mathsf{in}}), \mathsf{r}^{\mathsf{out}})$ :
1: $u = \mathsf{extend}(\mathsf{r}_1^{\mathsf{in}}, n)$
2: $w = \mathsf{extend}(\mathsf{r}_1^{\mathsf{in}}, n) \cdot \mathsf{r}_2^{\mathsf{in}} + \mathsf{r}^{\mathsf{out}}$
3: $z = 2 \cdot \mathsf{extend}(\mathsf{r}_1^{\mathsf{in}}, n) \cdot \mathsf{r}_2^{\mathsf{in}}$
4: share $(u, \mathsf{r}_2^{\mathsf{in}}, w, z)$
5: For $b \in \{0,1\}$, $k_b = u_b || \mathsf{r}_{2,b}^{\mathsf{in}} || w_b || z_b$

$\mathsf{Eval}_n^{\mathsf{select}}(b, k_b, (\hat{s}, \hat{x}))$ :
1: Parse $k_b$ as $u_b || \mathsf{r}_{2,b}^{\mathsf{in}} || w_b || z_b$
2: if $\hat{s} = 0$ then
3:     return $\hat{y}_b = u_b \cdot \hat{x} + w_b - z_b$
4: else
5:     return $\hat{y}_b = b \cdot \hat{x} - u_b \cdot \hat{x} - \mathsf{r}_{2,b}^{\mathsf{in}} + w_b$
6: end if
```

Fig. 1: Protocol for Select.

$\mathbb{U}_N$, and returns x if $s = 1$ and 0 otherwise. It is equivalent to unsigned mixed-bitwidth multiplication between x and s. That is, $\mathsf{select}_n(s, x) = s \cdot n$. Using the expression for offset function of unsigned mixed-bitwidth multiplication from [31], the offset function for select_n would be:

$$\begin{aligned}\mathsf{select}_n{}^{[(\mathsf{r}_1^{\mathsf{in}}, \mathsf{r}_2^{\mathsf{in}}), \mathsf{r}^{\mathsf{out}}]}(\hat{s}, \hat{x}) &= (\hat{s} - \mathsf{r}_1^{\mathsf{in}} + 2 \cdot 1\{\hat{s} < \mathsf{r}_1^{\mathsf{in}}\}) \cdot (\hat{x} - \mathsf{r}_2^{\mathsf{in}}) \\ &\quad + \mathsf{r}^{\mathsf{out}} \mod 2^n \\ &= \hat{s} \cdot \hat{x} - \mathsf{r}_1^{\mathsf{in}} \cdot \hat{x} - \hat{s} \cdot \mathsf{r}_2^{\mathsf{in}} + \mathsf{r}_1^{\mathsf{in}} \cdot \mathsf{r}_2^{\mathsf{in}} + \mathsf{r}^{\mathsf{out}} \\ &\quad + 2 \cdot 1\{\hat{s} = 0 \text{ and } \mathsf{r}_1^{\mathsf{in}} = 1\} \cdot (\hat{x} - \mathsf{r}_2^{\mathsf{in}}) \mod 2^n\end{aligned}$$

Here, we use the fact that $1\{\hat{s} < \mathsf{r}_1^{\mathsf{in}}\} = 1\{\hat{s} = 0 \text{ and } \mathsf{r}_1^{\mathsf{in}} = 1\}$ as $\hat{s}$ and $\mathsf{r}_1^{\mathsf{in}}$ are single bit values. Using this expression, we describe the protocol Π_n^{select} for select in Figure 1.

Theorem 2. Π_n^{select} *in Figure 1 realises* select_n *securely with* $\mathsf{keysize}(\Pi_n^{\mathsf{select}}) = 4n$ *and no communication.*

B. Signed Extension

[31] provides the following expression for the offset function of SignExt functionality (Section II):

$$\mathsf{SignExt}_{m,n}^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(\hat{x}) = \hat{x}' - \mathsf{r}^{\mathsf{in}} + 2^m \cdot 1\{\hat{x}' < \mathsf{r}^{\mathsf{in}}\} - 2^{m-1} + \mathsf{r}^{\mathsf{out}}$$

where $\hat{x}' = \hat{x} + 2^{m-1} \mod 2^m$. [31] also provides a non-interactive protocol for implementing it securely but suffers from a large key size (of $\mathsf{keysize}(\mathsf{DCF}_{m,n}) + n$). We provide a protocol with a smaller key size at the cost of an additional round and 2 bits of online communication. We describe the protocol $\Pi_{m,n}^{\mathsf{SignExt}}$ in Figure 2. In the protocol, we calculate the value of $1\{\hat{x}' < \mathsf{r}^{\mathsf{in}}\}$ as a one-bit masked value $\hat{w}$ in the first round with mask $r^{(w)}$. The dealer also sends shares of $(0, 2^m)$ or $(2^m, 0)$ depending on the mask $r^{(w)}$. Evaluators use $\hat{w}$ to select between 2^m and 0 as n-bit shares. As we only need the output of comparison as a single bit, we achieve a smaller key size compared to [31] due to a smaller DCF payload.

Theorem 3. $\Pi_{m,n}^{\mathsf{SignExt}}$ *in Figure 2 realizes* $\mathsf{SignExt}_{m,n}$ *securely such that* $\mathsf{keysize}(\Pi_{m,n}^{\mathsf{SignExt}}) = \mathsf{keysize}(\mathsf{DCF}_{m,1}) + 2n + 1$. *In the online phase, the protocol requires* 1 *evaluation of* $\mathsf{DCF}_{m,1}$ *and communication of* 2 *bits in* 1 *round.*

```
Signed Extension $\Pi_{m,n}^{\mathsf{SignExt}}$
$\mathsf{Gen}_{m,n}^{\mathsf{SignExt}}(\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$ :
1: $t = \mathsf{r}^{\mathsf{out}} - \mathsf{extend}(\mathsf{r}^{\mathsf{in}}, n) - 2^{m-1}$
2: $(k_0^<, k_1^<) \leftarrow \mathsf{Gen}_m^<(1^\lambda, \mathsf{r}^{\mathsf{in}}, 1, \mathbb{U}_2)$
3: $r^{(w)} \xleftarrow{\$} \mathbb{U}_2$
4: $\boldsymbol{p} = \{t, t + 2^m\} \ggg r^{(w)} \in \mathbb{U}_N^2$
5: share $(r^{(w)}, \boldsymbol{p})$
6: For $b \in \{0,1\}$, $k_b = k_b^< || r_b^{(w)} || \boldsymbol{p}_b$

$\mathsf{Eval}_{m,n}^{\mathsf{SignExt}}(b, k_b, \hat{x})$ :
1: Parse $k_b$ as $k_b^< || r_b^{(w)} || \boldsymbol{p}_b$
2: $\hat{w}_b \leftarrow \mathsf{Eval}_m^<(b, k_b^<, \hat{x}) + r_b^{(w)} \mod 2$
3: $\hat{w} = \mathsf{reconstruct}\ (\hat{w}_b)$
4: return $\hat{y}_b = b \cdot \hat{x} + p_{b,\hat{w}}$
```

Fig. 2: Protocol for SignExt.

C. Multiple Interval Containment

For public parameter arrays $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{U}_N^m$, multiple interval containment function, $\mathsf{MIC}_{n,m,\boldsymbol{p},\boldsymbol{q}} : \mathbb{U}_N \to \mathbb{U}_N^m$, calculates a vector $\boldsymbol{y} \in \mathbb{U}_N^m$ for a given input $x \in \mathbb{U}_N$, such that $\forall i \in [0, m-1]$, we have:

$$y_i = \begin{cases} 1 & \text{if } p_i \leqslant x \leqslant q_i \\ 0 & \text{otherwise} \end{cases}$$

[18] provides a protocol $\Pi_{n,m,\boldsymbol{p},\boldsymbol{q}}^{\mathsf{MIC}}$ for $\mathsf{MIC}_{n,m,\boldsymbol{p},\boldsymbol{q}}$. We omit the details of the protocol and use it as a black box in further protocols. We further restrict our discussion to the special case when $p_0 = 0$, $q_{m-1} = 2^n - 1$ and $p_i = q_{i-1} + 1 \forall i \in [1, m-1]$.

Theorem 4 (Multiple Interval Containment [18]). $\Pi_{n,m,\boldsymbol{p},\boldsymbol{q}}^{\mathsf{MIC}}$ *realizes* $\mathsf{MIC}_{n,m,\boldsymbol{p},\boldsymbol{q}}$ *securely such that* $\mathsf{keysize}(\Pi_{n,m,\boldsymbol{p},\boldsymbol{q}}^{\mathsf{MIC}}) = \mathsf{keysize}(\mathsf{DCF}_{n,n}) + mn$. *In the online phase, the protocol requires* m *evaluations of* $\mathsf{DCF}_{n,n}$.

V. Protocols for Secure Training

Training Functionalities. To implement secure training, we need protocols for the following functionalities: a) Linear layers, such as matrix multiplications and convolutions; and b) Activation functions, such as ReLU and Maxpool. Linear layers are computed securely using the same method from LLAMA [31] (i.e., through the use of Beaver triples that are provided as the key by the dealer). In fixed-point arithmetic, multiplications must be followed by a truncate operation in order to maintain scale. That is, multiplying two fixed-point values with scale f over integers, results in a fixed-point value with implicit scale as $2f$ and hence, we need a truncation by f to obtain result with scale f. The literature considers three kinds of truncations when implementing fixed-point arithmetic in secure computation: faithful truncation or arithmetic right shift, stochastic truncation [32], [33], [39], and local truncations [47]. Most prior works on secure training [47], [58],

[60], [64] use local truncations as they are most efficient due to being local operations. LLAMA [31] provides an FSS protocol for faithful truncation. Inspired by the work of Gupta et al. [32] and Keller and Sun [39], we provide FSS protocols for stochastic truncation (defined in Section V-B) that has been observed to result in better training accuracy. In ML inference and training, linear layers are followed by activation functions. When implementing fixed-point training, this would correspond to linear layers being followed by a truncation, which is then followed by ReLU and sometimes Maxpool. In such cases, we observe that it is more efficient to fuse and compute these nodes together (instead of computing truncation, ReLU and Maxpool separately).

Section Overview. We begin by describing our new protocols for ReLU (Section V-A) and stochastic truncation (Section V-B) with reduced key size. Next, in Section V-C, we show how to fuse the stochastic truncate nodes with activation functions such as ReLU and ReLU+Maxpool to obtain an even lower key size and compute compared to the naive approach of sequential computation. Throughout, our focus is on reducing the key size (even if we sometimes pay slightly in terms of number of online rounds of communication).

A. ReLU

For a signed value x, the ReLU functionality returns $\max(x, 0)$. So, when $x \in \mathbb{U}_N$ is an n-bit 2's-complement representation of an underlying signed value, the ReLU functionality is equivalent to:

$$\mathsf{ReLU}_n(x) = x \cdot 1\{x < 2^{n-1}\}$$

$1\{x < 2^{n-1}\}$ is also called the Derivative of ReLU or DReLU.

$$\mathsf{DReLU}_n(x) = 1\{x < 2^{n-1}\}$$
$$\mathsf{ReLU}_n(x) = \mathsf{select}(\mathsf{DReLU}_n(x), x)$$

AriaNN [55] provides a 1-round protocol for ReLU_n (with a 1-bit error) with a key size of $\approx (n+1)(\lambda+2n)$, while [18] constructed a non-interactive protocol (with no error) with the same key size. Here, we construct a 1-round ReLU_n protocol (with no error) with a key size of $n(\lambda+7)+\lambda+2$. We do this by first calculating $\mathsf{DReLU}_n(x)$ in the first round with a single bit output and then using the protocol Π_n^{select} to output x or 0 based on the comparison output. Since we only need the comparison output as a single bit, the key size of this protocol is smaller than the spline-based protocol in [18]. Concretely, for $n = 64$, this results in a $\approx 2\times$ reduction in key size at the cost of one additional round and 2 additional bits of online communication compared to [18].

We use the following expression for the offset function of DReLU_n (proved in Appendix E): For $\hat{y} = \hat{x} + 2^{n-1} \mod 2^n$

$$\mathsf{DReLU}_n^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}(\hat{x}) = 1\{\hat{y} < \mathsf{r}^{\mathsf{in}}\} - 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} + 1\{\hat{y} \geq 2^{n-1}\} + \mathsf{r}^{\mathsf{out}} \mod 2$$

Based on this, we describe the protocol for DReLU_n, Π_n^{DReLU}, in Figure 3. Protocol for ReLU_n, Π_n^{ReLU}, is obtained by running Π_n^{DReLU} followed by a round of reconstruction for $\hat{y}$ and Π_n^{select}.

```
**DReLU** $\Pi_n^{\mathsf{DReLU}}$
$\mathsf{Gen}_n^{\mathsf{DReLU}}(\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$ :
  1: $(k_0^<, k_1^<) \leftarrow \mathsf{Gen}_n^<(1^\lambda, \mathsf{r}^{\mathsf{in}}, 1, \mathbb{U}_2)$
  2: share $\mathsf{r}^{\mathsf{out}}$
  3: For $b \in \{0,1\}$, $k_b = \mathsf{r}_b^{\mathsf{out}} || k_b^<$

$\mathsf{Eval}_n^{\mathsf{DReLU}}(b, k_b, \hat{x})$ :
  1: Parse $k_b$ as $\mathsf{r}_b^{\mathsf{out}} || k_b^<$
  2: $\hat{y} = \hat{x} + 2^{n-1} \mod 2^n$
  3: $u_b \leftarrow \mathsf{Eval}_n^<(b, k_b^<, \hat{x})$
  4: $v_b \leftarrow \mathsf{Eval}_n^<(b, k_b^<, \hat{y})$
  5: **return** $\hat{y}_b = v_b - u_b + b \cdot 1\{\hat{y} \geqslant 2^{n-1}\} + \mathsf{r}_b^{\mathsf{out}} \mod 2$
```

Fig. 3: Protocol for DReLU.

Theorem 5. *Π_n^{ReLU} realizes ReLU_n securely such that $\mathsf{keysize}(\Pi_n^{\mathsf{ReLU}}) = \mathsf{keysize}(\mathsf{DCF}_{n,1}) + \mathsf{keysize}(\Pi_n^{\mathsf{select}}) + 1$. In the online phase, the protocol requires 2 evaluations of $\mathsf{DCF}_{n,1}$ and communication of 2 bits in 1 round.*

B. Stochastic Truncation

Gupta et al. [32] demonstrated the importance of *stochastic-truncation* (as a replacement for round-to-nearest truncation) for machine learning training with limited precision. In this technique, the result of truncation is rounded up or down, with a probability depending on the truncated fractional part.

Definition 3. *Let $x \in \mathbb{U}_N$ be an n-bit number. The stochastic-truncation of x with f, represented by $y = x \gg_{\mathsf{st}} f$, is an n-bit number $y \in \mathbb{U}_N$ such that:*

$$y = \begin{cases} (x \gg_A f) & \textit{with probability} \quad 1 - t \cdot 2^{-f} \\ (x \gg_A f) + 1 & \textit{with probability} \quad t \cdot 2^{-f} \end{cases}$$

where $t = x \mod 2^f$.

We prove the following lemma (proof provided in Appendix D) that allows us to securely realize stochastic truncation at the cost of signed extension from $n - f$ to n bits.

Lemma 1. *Consider a masked value $\hat{x} \in \mathbb{U}_N$ with underlying value x and random mask $r^{(x)}$. Let $\hat{y} = \mathsf{TR}(\hat{x}, f)$ and $r^{(y)} = \mathsf{TR}(r^{(x)}, f)$. Then the following holds:*

$$x \gg_{\mathsf{st}} f = \mathsf{SignExt}_{n-f,n}(\hat{y} - r^{(y)})$$

Protocol. As a result of this lemma, the evaluators and dealer can locally truncate-reduce $\hat{x}$ and $r^{(x)}$ to get $\hat{y}$ and $r^{(y)}$ respectively. Then, by setting the input masked-value to $\hat{y}$ and the input mask to $r^{(y)}$, the protocol $\Pi_{n-f,n}^{\mathsf{SignExt}}$ can be used to extend the value to n bits. We denote the protocol by $\Pi_{n,f}^{\mathsf{stTr}}$.

Theorem 6. *$\Pi_{n,f}^{\mathsf{stTr}}$ realizes stochastic-truncation by f securely such that $\mathsf{keysize}(\Pi_{n,f}^{\mathsf{stTr}}) = \mathsf{keysize}(\Pi_{n-f,n}^{\mathsf{SignExt}})$. In the online phase, the protocol requires 1 evaluation of $\Pi_{n-f,n}^{\mathsf{SignExt}}$ and communication of 2 bits in 1 round.*

LLAMA [31] had an implementation for stochastic truncation with key size $\mathsf{keysize}(\mathsf{DCF}_{n-1,2n}) + 3n$ using relation from [33]. For $n = 64, f = 24$, our key size is $3\times$ lower.

C. Stochastic-Truncation + Activations

As discussed earlier, when linear layers are followed by activations such as ReLU and Maxpool, we fuse the stochastic truncation node along with ReLU (or ReLU and Maxpool, depending on the nodes present) to obtain a single protocol for the fused functionality. In this section, we describe how this is done for the case of ReLU; in Appendix F, we describe the case of ReLU+MaxPool.

As mentioned before, linear layer is computed over $\mathbb{U}_N$ followed by stochastic truncation by a public scale f. We define a fused functionality, *stochastic-truncation + ReLU*, which takes as input $x \in \mathbb{U}_N$, stochastically-truncates it by f, and returns the ReLU over the truncated value. Formally,

$$\mathsf{stTrReLU}_{n,f}(x) = \mathsf{ReLU}_n(x \gg_{\mathsf{st}} f)$$

The above expression can be realized in 2-rounds by running the protocol $\hat{\Pi}^{\mathsf{stTr}}_{n,f}$ followed by Π^{ReLU}_n. We improve over this significantly in 2 steps described below.

First, based on Lemma 1, in stochastic truncation, the dealer and the evaluators do truncate-reduce of mask and masked input to $(n-f)$-bits, followed by a signed extension to n-bits. In the above protocol, this is followed by a ReLU computation on n-bit inputs. We observe that we can switch the order of ReLU and extension that allows us to compute ReLU on $(n-f)$ bits instead of n-bits, reducing the key size and online compute as well. Moreover, since ReLU output is always non-negative, we can use zero extension, whose protocol is very similar to signed extension (Section IV-B).

Second, we improve upon above by providing a new protocol that does ReLU and zero extension together by leveraging similar comparisons done for DReLU and ZeroExt. We call this functionality ReLU-Extend, denoted by $\mathsf{ReLUExt}_{n-f,n}(x) = \mathsf{ZeroExt}_{n-f,n}(\mathsf{ReLU}_{n-f}(x))$.

For a value $x \in \mathbb{U}_{2^{n-f}}$, mask r^{in} and masked value $\hat{x}$, let $d = \mathsf{DReLU}(x)$ and $w = 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\}$. Moeover, from [31],

$$\mathsf{ZeroExt}^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{n-f,n}(\hat{x}) = \hat{x} - \mathsf{r}^{\mathsf{in}} + 2^{n-f} \cdot w + \mathsf{r}^{\mathsf{out}}$$

Then, the offset gate for $\mathsf{ReLUExt}_{n-f,n}$ can be written as

$$\mathsf{ReLUExt}^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{n-f,n}(\hat{x}) = \begin{cases} \underline{0} + \mathsf{r}^{\mathsf{out}} & d = 0, w = 0 \\ \underline{0} + \mathsf{r}^{\mathsf{out}} & d = 0, w = 1 \\ \underline{\hat{x}} + \mathsf{r}^{\mathsf{out}} - \mathsf{r}^{\mathsf{in}} & d = 1, w = 0 \\ \underline{\hat{x} + 2^{n-f}} + \mathsf{r}^{\mathsf{out}} - \mathsf{r}^{\mathsf{in}} & d = 1, w = 1 \end{cases}$$

That is, we need to compute a 1-out-of-4 selection based on values of d and w. Now, both d and w use comparisons with r^{in} and hence can be computed using a single DCF key and their masked value can be obtained in a single round of interaction. Now the selection can be done using 2 consecutive calls to select, resulting in an overall 2 round protocol. We improve this further to 1 round below.

We compute d, w over $\mathbb{U}_4$ instead of $\mathbb{U}_2$. Let $i = 2 \cdot d + w \in \mathbb{U}_4$ be the index of this 1-out-of-4 selection. We denote the masked value and the secret mask of i by $\hat{i}$ and $r^{(i)}$, respectively. Note that the underlined values in the above expression are known to both parties. To obliviously select between these 4 values, the dealer gives out shares of an array $\boldsymbol{p} \in \mathbb{U}_N^4$ such that $p_k = 1$ when $k = 4 - r^{(i)}$ and 0 otherwise. In the online phase, evaluators rotate $\boldsymbol{p}_b$ to the right by $\hat{i}$ places to get shares of a one-hot array which is 1 at position i. Inner-product of this array with $\{0, 0, \hat{x}, \hat{x} + 2^{n-f}\}$ produces shares of the selected underlined value.

Based on the value of d, the evaluators need to obliviously select between $\mathsf{r}^{\mathsf{out}}$ and $\mathsf{r}^{\mathsf{out}} - \mathsf{r}^{\mathsf{in}}$, where both values are known to the dealer. To do this, the dealer gives out shares of an array $\boldsymbol{q} \in \mathbb{U}_N^2$ with elements $\mathsf{r}^{\mathsf{out}}$ and $\mathsf{r}^{\mathsf{out}} - \mathsf{r}^{\mathsf{in}}$, swapped when $r^{(d)} \bmod 2 = 1$. In the online phase, evaluators index the array $\boldsymbol{q}_b$ at $\hat{d} \mod 2$. Adding the two selected shares results in shares of the required masked output.

We present the protocol $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$ for $\mathsf{ReLUExt}_{n-f,n}$ in Figure 4 and proof of its security in Appendix G. Using this, we can trivially obtain a protocol for $\mathsf{stTrReLU}_{n,f}$ by locally truncate-reducing the input, followed by the protocol $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$.

Theorem 7. *$\Pi^{\mathsf{stTrReLU}}_{n,f}$ realizes $\mathsf{stTrReLU}_{n,f}$ securely such that $\mathsf{keysize}(\Pi^{\mathsf{stTrReLU}}_{n,f}) = \mathsf{keysize}(\mathsf{DCF}_{n-f,2}) + 6n + 4$. In the online phase, the protocol requires 2 evaluations of $\mathsf{DCF}_{n-f,2}$ and communication of 8 bits in 1 round.*

In contrast, the naive way of implementation using stochastic truncation followed by ReLU requires a key size of $\mathsf{keysize}(\mathsf{DCF}_{n-1,2n}) + \mathsf{keysize}(\mathsf{DCF}_{n,2n}) + 8n$ in LLAMA [31] and key size of $\mathsf{keysize}(\mathsf{DCF}_{n-f,1}) + \mathsf{keysize}(\mathsf{DCF}_{n,1}) + 6n + 2$ using our protocols. Hence, for $n = 64, f = 24$ our key size is $5.8\times$ and $2.5\times$ lower, respectively. Over LLAMA, we also have $2.4\times$ lower AES evaluations in Eval ($6n$ vs $4(n-f)$).

VI. Protocols for Softmax

As discussed earlier, to mirror the exact computation performed in [32], we calculate softmax accurately in floating-point. To do this securely, we make use of the state-of-the-art floating-point computations library, SecFloat [51]. However, in order for the parties to invoke the softmax protocol from this library, they must hold secret shares of the floating-point representation of the input. Hence, we require a protocol that would convert masked fixed-point values (from an FSS scheme) into secret shares of the corresponding floating-point values according to the representation of [51]. Similarly, in order to use the output of the softmax computation in the rest of the training protocol, we require a protocol that would convert secret shares of floating-point values back to the corresponding (masked) fixed-point values. We begin with background on floating-point representations and softmax followed by our protocols for FixToFloat and FloatToFix in Section VI-A and Section VI-B.

Floating-point representation. Similar to SecFloat [51], we represent a 32-bit floating-point number α using four values

```
ReLU-Extend $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$
$\mathsf{Gen}^{\mathsf{ReLUExt}}_{n-f,n}(\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$ :
  1: $(k_0^<, k_1^<) \leftarrow \mathsf{Gen}^<_{n-f}(1^\lambda, \mathsf{r}^{\mathsf{in}}, 1, \mathbb{U}_4)$
  2: $r^{(d)} \xleftarrow{\$} \mathbb{U}_4$
  3: $r^{(w)} \xleftarrow{\$} \mathbb{U}_4$
  4: $r^{(i)} = 2 \cdot r^{(d)} + r^{(w)}$
  5: $\boldsymbol{p} = \{1, 0, 0, 0\} \lll r^{(i)} \in \mathbb{U}_N^4$
  6: $\boldsymbol{q} = \{\mathsf{r}^{\mathsf{out}}, \mathsf{r}^{\mathsf{out}} - \mathsf{r}^{\mathsf{in}}\} \ggg (r^{(d)} \mod 2) \in \mathbb{U}_N^2$
  7: share $(r^{(d)}, r^{(w)}, \boldsymbol{p}, \boldsymbol{q})$
  8: For $b \in \{0,1\}, k_b = k_b^< || r_b^{(d)} || r_b^{(w)} || \boldsymbol{p}_b || \boldsymbol{q}_b$

$\mathsf{Eval}^{\mathsf{ReLUExt}}_{n-f,n}(b, k_b, \hat{x})$ :
  1: Parse $k_b$ as $k_b^< || r_b^{(d)} || r_b^{(w)} || \boldsymbol{p}_b || \boldsymbol{q}_b$
  2: $\hat{y} = \hat{x} + 2^{n-f-1} \mod 2^{n-f}$
  3: $w_b \leftarrow \mathsf{Eval}^<_{n-f}(b, k_b^<, \hat{x})$
  4: $\hat{w}_b \leftarrow w_b + r_b^{(w)} \mod 4$
  5: $\hat{d}_b \leftarrow \mathsf{Eval}^<_{n-f}(b, k_b^<, \hat{y}) - w_b + b \cdot 1\{\hat{y} \geqslant 2^{n-f-1}\} + r_b^{(d)} \mod 4$
  6: $(\hat{w}, \hat{d}) = $ reconstruct $(\hat{w}_b, \hat{d}_b)$
  7: $\hat{i} = 2 \cdot \hat{d} + \hat{w} \mod 4$
  8: $\boldsymbol{p}_b' = \boldsymbol{p}_b \ggg \hat{i}$
  9: $\hat{j} = \hat{d} \mod 2$
  10: return $\hat{u}_b = p'_{b,3} \cdot (\hat{x} + 2^{n-f}) + p'_{b,2} \cdot \hat{x} + q_{b,\hat{j}} \mod N$
```

Fig. 4: Protocol for ReLUExt.

$(z, s, e, m) \in \mathbb{FP}$ where $z \in \{0,1\}$ represents zero-bit (set when $\alpha = 0$), $s \in \{0,1\}$ represents the sign bit (set when $\alpha < 0$), $e \in \mathbb{U}_{2^{10}}$ represents the unbiased signed exponent with values lying in the range $[-127, 128]$ and $m \in \mathbb{U}_{2^{24}}$ represents normalized unsigned fixed-point mantissa with scale 23, taking values in the range $[2^{23}, 2^{24} - 1] \cup \{0\}$. Moreover, $\alpha = (z, s, e, m)$ represents the real number $(1-z) \cdot (1-2s) \cdot 2^{\mathsf{int}_{10}(e)} \cdot [\![m]\!]^+_{24,23}$. When $\alpha = 0$, $e = -126$ and $m = 0$ holds.

Softmax. For a d-dimensional vector $\boldsymbol{x} \in \mathbb{R}^d$ of real numbers, softmax calculates a vector $\boldsymbol{y} \in \mathbb{R}^d$ such that:

$$\forall i \in [0, d-1],\ y_i = \frac{e^{x_i}}{\sum_{i=0}^{d-1} e^{x_i}} = \frac{e^{x_i - x_{\max}}}{\sum_{i=0}^{d-1} e^{x_i - x_{\max}}} \tag{1}$$

where $x_{\max} = \max(x_0, x_1, \ldots x_{d-1})$. The latter expression is usually preferred as exponentials in the former expression can become arbitrarily large leading to overflows. The second expression on the other hand limits the exponential outputs to lie in the range $(0, 1]$.

A. FixToFloat

To convert a fixed point number $x \in \mathbb{U}_N$ with scale f to the equivalent floating-point number $(z, s, e, m) \in \mathbb{FP}$, we have to find (z, s, e, m) such that the underlying real values are close. So, the following relation[2] should hold:

$$[\![x]\!]_{n,f} = (1-z) \cdot (1-2s) \cdot 2^{\mathsf{int}_{10}(e)} \cdot [\![m]\!]^+_{24,23}$$
$$\frac{\mathsf{int}_n(x)}{2^f} = (1-z) \cdot (1-2s) \cdot 2^{\mathsf{int}_{10}(e)} \cdot \frac{\mathsf{uint}_{24}(m)}{2^{23}}$$

As z and s denote the zero and sign bit respectively, calculating them is trivial using the following relations:

$$z = 1\{x = 0\}$$
$$s = 1\{x \geqslant 2^{n-1}\}$$

Note that when $z = 1$, $e = -126$ and $m = 0$ holds. In the case when $z = 0$, it only remains to find e and m such that:

$$\left|\frac{\mathsf{int}_n(x)}{2^f}\right| = 2^{\mathsf{int}_{10}(e)} \cdot \frac{\mathsf{uint}_{24}(m)}{2^{23}}$$
$$\implies |\mathsf{int}_n(x)| = 2^{\mathsf{int}_{10}(e+f-23)} \cdot \mathsf{uint}_{24}(m)$$

Computing e. For a given x, there can be multiple solutions to this equation. However, as m has to be normalized, i.e., lies in the range $[2^{23}, 2^{24} - 1]$, there is a unique pair (m, e) which satisfies both these constraints. Let $k \leqslant n$ be a number such that $2^{k-1} \leqslant |\mathsf{int}_n(x)| < 2^k$. Then, $2^{23} \leqslant 2^{24-k} \cdot |\mathsf{int}_n(x)| < 2^{24}$. Multiplying 2^{24-k} to both sides of the above equation,

$$2^{24-k} \cdot |\mathsf{int}_n(x)| = 2^{\mathsf{int}_{10}(e+f+1-k)} \cdot \mathsf{uint}_{24}(m)$$

As the LHS and $\mathsf{uint}_{24}(m)$ both lie in the range $[2^{23}, 2^{24} - 1]$, the above equation can only hold when:

$$2^{\mathsf{int}_{10}(e+f+1-k)} = 1 \implies e = k - f - 1$$

Computing m. Now we have that:

$$\mathsf{uint}_{24}(m) = 2^{24-k} \cdot |\mathsf{int}_n(x)|$$

Note that $|\mathsf{int}_n(x)|$ can be calculated as an n-bit number using the relation $|x| = 2 \cdot \mathsf{ReLU}_n(x) - x$. To calculate m, we have:

$$\mathsf{uint}_{24}(m) = 2^{24-k} \cdot |\mathsf{int}_n(x)| = \frac{2^{n-k} \cdot |\mathsf{int}_n(x)|}{2^{n-24}}$$

In the fraction above, we notice that as $|\mathsf{int}_n(x)| < 2^k$, the numerator $2^{n-k} \cdot |\mathsf{int}_n(x)| < 2^n$ can be represented accurately as an n-bit unsigned number. As we need to calculate m as a 24-bit number, it suffices to truncate-reduce the numerator by $n - 24$ to get an accurate approximation of m.

Putting things together, for a given n-bit fixed-point number x with precision f, we define functionality $\mathsf{FixToFloat}_{n,f} : \mathbb{U}_N \to \mathbb{FP}$ which calculates the floating-point number $(z, s, e, m) \in \mathbb{FP}$ where:

$$z = 1\{x = 0\};\ s = 1\{x \geqslant 2^{n-1}\}$$
$$e = \begin{cases} k - f - 1 & \text{if } x \neq 0 \\ -126 & \text{if } x = 0 \end{cases}$$
$$m = \mathsf{TR}(|x| \cdot 2^{n-k}, n - 24)$$

[2]For reals a and b, we abuse $a = b$ to mean $|a - b| < 2^{-23}$.

with $|x|$ and k defined similarly as above. A natural way to compute $\mathsf{FixToFloat}_{n,f}$ securely is by using the existing protocols for Zero-Test [21], DReLU, multiple interval containment (to compute 2^{n-k}) and ReLU. However, each of these protocols would require a DCF key (DPF key, in case of Zero-Test) and hence, the overall protocol would incur a key size of $\approx n(4\lambda + 3n + 15) + 4\lambda + 3$ bits. We describe an alternate approach to achieve the same result using a single DCF key and a key size of $n(\lambda + 3n + 9) + \lambda + 2$ bits.

Let $\boldsymbol{p}^{(n)}$ and $\boldsymbol{q}^{(n)}$ be two constant series' of length $2n$, representing $2n$ disjoint intervals in $\mathbb{U}_N$, such that:

$$p_i^{(n)} = \begin{cases} 0 & \text{if } i = 0 \\ 2^{i-2} & \text{if } i \in [1, n-1] \\ 2^{n-1} & \text{if } i = n \\ 2^n - 2^{2n-i+1} + 1 & \text{if } i \in [n+1, 2n-1] \end{cases}$$

$$q_i^{(n)} = \begin{cases} 2^n - 1 & \text{if } i = 2n-1 \\ p_{i+1}^{(n)} - 1 & \text{otherwise} \end{cases}$$

Let $u = 2^{n-k}$. We observe that for all values of x lying in the interval $[p_i^{(n)}, q_i^{(n)}]$, u, e, z, and s hold constant values. Hence, we can use the protocol for multiple interval containment (Section IV-C) to get shares of a one-hot vector that specifies which interval x belongs to. We can now get shares of values u, e, z, and s by elementwise-multiplying the shares of the vector with the correct constant value in the corresponding interval and adding them. Then, we can calculate $|x| = 2\mathsf{ReLU}_n(x) - x$ cheaply by reusing the sign bit. The resulting value when multiplied with u and truncate-reduced by $(n-24)$, gives mantissa m.

We provide the formal protocol $\Pi_{n,f}^{\mathsf{FixToFloat}}$ and its security proof in Appendix H with cost summarized below.

Theorem 8. $\Pi_{n,f}^{\mathsf{FixToFloat}}$ *realizes* $\mathsf{FixToFloat}_{n,f}$ *securely such that* $\mathsf{keysize}(\Pi_{n,f}^{\mathsf{FixToFloat}}) = \mathsf{keysize}(\Pi_{n,2n,\boldsymbol{p}^{(n)},\boldsymbol{q}^{(n)}}^{\mathsf{MIC}}) + \mathsf{keysize}(\Pi_n^{\mathsf{select}}) + 3n + 1$. *In the online phase, the protocol requires* 1 *evaluation of* $\Pi_{n,2n,\boldsymbol{p}^{(n)},\boldsymbol{q}^{(n)}}^{\mathsf{MIC}}$ *and costs communication of* $4n + 2$ *bits in* 2 *rounds.*

B. FloatToFix

Given a secret-shared floating-point value $(z, s, e, m) \in \mathbb{FP}$, we need to compute shares of one of the two closest n-bit fixed-point number x with scale f. Our protocol $\Pi_{n,f}^{\mathsf{FloatToFix}}$ that realizes the above functionality $\mathsf{FloatToFix}_{n,f}$ for softmax outputs uses similar ideas as above and we delegate details to Appendix I. The following theorem summarizes its cost.

Theorem 9. $\Pi_{n,f}^{\mathsf{FloatToFix}}$ *realizes* $\mathsf{FloatToFix}_{n,f}$ *securely such that* $\mathsf{keysize}(\Pi_{n,f}^{\mathsf{FloatToFix}}) = \mathsf{keysize}(\mathsf{DCF}_{24,\mathbb{U}_2}) + \mathsf{keysize}(\Pi_n^{\mathsf{select}}) + 2049n + 35$. *In the online phase, the protocol requires* 1 *evaluation of* $\mathsf{DCF}_{24,\mathbb{U}_2}$ *and costs communication of* $2n + 70$ *bits in* 2 *rounds.*

C. End-to-end training

As discussed earlier in Section V, individual FSS-based protocols for convolution, matrix multiplications, ReLU, Maxpool and so on can be combined to build end-to-end protocols. When convolutions (or matrix multiplications) are followed by ReLU (or ReLU+Maxpool), they are replaced by the protocols for convolutions (correspondingly matrix multiplications) followed by $\Pi_{n,f}^{\mathsf{stTrReLU}}$ (or $\Pi_{n,f}^{\mathsf{TRM}}$ respectively). Training protocols require softmax computation at the end of the forward pass and here we first convert masked fixed-point outputs to secret-shared floating point numbers using $\Pi_{n,f}^{\mathsf{FixToFloat}}$, call the protocol for softmax from SECFLOAT [51], and then finally convert the secret-shared floating point numbers back to masked fixed-point numbers using $\Pi_{n,f}^{\mathsf{FloatToFix}}$. To compute the backward pass for ReLU and Maxpool, we re-use the comparison outputs from the forward pass and combine them with Π_n^{select} and bitwise-AND; hence, there is no benefit in fusing nodes here. The security of the end-to-end protocol can be argued in the simulation paradigm. For details, see Appendix J.

VII. IMPLEMENTATION

We implement ORCA as a C++ library for easy use. It contains optimized GPU kernels for the FSS gates described in Sections IV and V. ORCA's software framework supports the key functionalities necessary to implement GPU-optimized FSS protocols. We leverage optimized kernels from NVIDIA's CUDA libraries where available and write our own CUDA kernel where necessary. Unlike a prior work CryptGPU [58], we do *not* embed 64-bit integers into 64-bit floating-point numbers for leveraging optimized floating-point kernels from NVIDIA's cuBLAS [1] and cuDNN [7] libraries. Instead, we stick to integer kernels to avoid the overheads of embedding, as did the prior work of PIRANHA [64].

Table II lists the key functionalities that our framework implements as GPU kernels. It also reports which ones are written by us (ORCA). We use well-optimized CUDA kernels from NVIDIA's CUTLASS [5] for convolutions and matrix multiplications in the linear layers. The rest of the optimized kernels were implemented as part of this work. In Section III, we described various optimization that we perform (AES, LAYOUT, MEM) to speed up DCF. We apply several of the same GPU-centric optimizations to other kernels as suitable. The table lists optimizations applicable to each kernel.

Some parts of our framework run on the CPU. As mentioned in Section VI, we use SECFLOAT [12] to compute softmax in floating point on the CPU. We did not accelerate the protocols for softmax with GPUs since we find that for larger models (Table IX, Appendix A) we are bottlenecked by key read and not softmax. Furthermore, to go from fixed-point to floating point and back, we implement the protocols FixToFloat and FloatToFix outlined in Figures 7 and 8 on the CPU as these conversions have tiny overheads.

Beyond the design optimizations mentioned in Section III, we also ensure optimized implementation of the software stack through several practical considerations as follows. For every invocation of an FSS function (e.g., a DCF) on the GPU, GPU memory needs to be allocated to hold the function's key. Same needs to be de-allocated after the completion of the function. The repeated allocation/de-allocation of GPU

Func.	Description	Source	Optimizations
Matrix mult.	Multiplies two matrices	CUTLASS	–
Conv	Performs convolution	CUTLASS	–
DCF	The DCF described in [18]	ORCA	AES, LAYOUT, MEM
DReLU	The DReLU protocol in Figure 3	ORCA	AES, LAYOUT, MEM
Select	The Select protocol in Figure 1 and its variants in Figures 2 and 4	ORCA	LAYOUT, MEM
Bitwise AND	Securely computes AND of two bits	ORCA	LAYOUT, MEM

Table II: Key components of accelerated FSS

memory adds overheads. Toward this, we leverage a new feature introduced in a recent CUDA release (CUDA 11.2), called CUDA memory pools [6], to reserve GPU memory for fast allocation/de-allocation of keys. Further, we pre-allocate host-side communication buffers on the CPU to avoid overheads of dynamic memory allocations at runtime. We also pin host memory on the CPU, allowing for faster data transfers between GPU and CPU using DMA [9]. We use multiple CPU threads to overlap the CPU tasks of reading keys from the SSD to the CPU DRAM, launching GPU kernels, and to communicate with the other party.

Besides accelerating FSS evaluation, we also accelerate the FSS dealer. We use NVIDIA's cuRAND library [4] to generate the randomness on the GPU for the dealer. We will opensource the framework along with the publication of this work.

Further, we extended LLAMA [31], the state-of-the-art tool for FSS-based ML on CPUs, to support training by combining its protocols for inference with our softmax and call its protocols for operations occurring in training. We name this LLAMA extended to support training as LLAMA in the sequel and use it to show our systems and cryptographic contributions quantitatively in Section VIII-C.

VIII. EVALUATION

We compare ORCA against the state-of-the-art secure 2PC training tools that support the same threat model as ours, i.e., a trusted dealer provides correlated randomness to two parties in an offline phase, and then the two parties run a 2PC protocol for an ML task on their sensitive data. We provide an empirical evaluation to justify the following claims:

- ORCA faithfully implements the quantized training algorithms in ML literature [32] that mix floating-point and fixed-point arithmetic (Figure 5). ORCA matches the accuracy of PIRANHA (the current state-of-the-art in accelerating secure training with GPUs) generated models in $4-19\times$ less time and with $51-123\times$ less communication (Table III).
- On identical training and inference tasks, ORCA outperforms (GPU-based) PIRANHA by up to $9\times$ in latency and up to $14\times$ in communication (Table IV). ORCA also outperforms state-of-the-art CPU-based secure training [39] by up to $441\times$ (Table V).
- ORCA's GPU-based protocols are up to $29\times$ more efficient than their CPU counterparts (Table VI). Moreover, the size of FSS keys required by ORCA's protocols is up to $5\times$ lower (see Table VII) than LLAMA, the state-of-the-art tool for FSS-based ML.
- ORCA enables sub-second ImageNet-scale inference (Section VIII-C).

Parameter setting: We evaluate ORCA and PIRANHA on a fixed-point representation with bitwidth $n = 64$ and precision $f = 24$. The CPU-baseline [39] also uses a bitwidth of 64.

Datasets and models: We use the same datasets as PIRANHA for our training evaluation, i.e., 10-class MNIST and CIFAR. The training set of MNIST has 60,000 monochromatic 28×28 images and the test set has 10,000 images. The training set of CIFAR has 50,000 RGB images and the test set has 10,000 images. We annotate the names of models for MNIST and CIFAR with subscripts M and C, respectively. PIRANHA [64] uses the following models: P-SecureML$_{\mathsf{M}}$ and P-Lenet$_{\mathsf{M}}$ for MNIST, and P-AlexNet$_{\mathsf{C}}$ and P-VGG16$_{\mathsf{C}}$ for CIFAR. P-VGG16$_{\mathsf{C}}$ has over 100 million parameters and is the largest model of PIRANHA. These models use approximations, e.g., local truncations, that are not used by state-of-the-art CPU-based secure training tools. To compare against them we use the following models in [39]: Model-B$_{\mathsf{M}}$ [39] and AlexNet$_{\mathsf{C}}$ [58]. For small models, we use the 36k parameter CNN2$_{\mathsf{M}}$ model with 2 convolutions and the 200k parameter CNN3$_{\mathsf{C}}$ model with 3 convolutions from Gupta et al. [32].

Finally, we also do a preliminary evaluation on VGG16$_I$ [14], [57] for ImageNet-1000 [27] dataset. This model is much larger than the other models as it operates on 224×224 RGB images, which are 7×7 bigger than CIFAR images. In particular, compared to P-VGG16$_{\mathsf{C}}$, the largest model in PIRANHA, VGG16$_I$ has $49\times$ more multiplications and $73\times$ more comparisons. It also has $6\times$ the parameters of ResNet50 for ImageNet-1000, a commonly used model in prior secure inference works [31], [33], [53], [58]. We evaluate secure VGG16$_I$ inference with maxpools and stochastic truncations to match its floating-point accuracy. We omit its secure training as it is currently impractical; even with ORCA's speedups, end-to-end training on ImageNet-1000 will take years.

Evaluation setup: We perform our experiments on two virtual machines, connected in a LAN setup with 9.4 Gbps bandwidth and 0.05 ms RTT, each equipped with an NVIDIA RTX A6000 GPU with 42GB of onboard memory and an AMD Epyc 7742 processor. Each machine sports nearly a TB of RAM and a Micron 7300 NVMe SSD connected with a PCIe3 bus. The SSD supports 2.9GBps of sequential read bandwidth. We use four of the CPU cores for CPU-only experiments and use three cores for the GPU experiments.

The training times reported for ORCA are inclusive of the time required to move FSS keys from SSD to RAM, as keys required in training for many iterations over many epochs will not fit in RAM. This is not an issue for inference and the inference time measurements assume that the keys reside in RAM. Note that the time measurements of inference/training for the baselines don't include the time to load keys or pre-

	Accuracy		Time (in min)		Comm. (in GB)	
Dataset	PIRANHA	ORCA	PIRANHA	ORCA	PIRANHA	ORCA
MNIST	96.8 (−0.3%)	97.1	56 (4×)	14	2,168.4 (51×)	42.9
CIFAR-10	55 (−4%)	59	1179 (19×)	63	65231.3 (123×)	532.5
	55 (−15%)	70	1179 (7.5×)	158	65231.3 (49×)	1331.3

Table III: ORCA matches accuracy of PIRANHA in lesser time on MNIST and CIFAR-10.

processing material into RAM. Similar to [64], for all the baselines and ORCA, we only report the online time[3].

A. *End-to-end training with* ORCA

Our goal is to show that for a given classification task, ORCA can train a model that matches the accuracy of the models trained by PIRANHA, while incurring significantly less time and communication. Recall that PIRANHA's implementation of softmax leaks private information [62], while ORCA, with its floating-point softmax, provides end-to-end security.

On the MNIST dataset, PIRANHA reports an accuracy of 96.8% while training P-LeNet$_\text{M}$, incurring about an hour in training time and about 2TB of communication. Table III shows that ORCA matches this accuracy with 4× less time and 51× less communication while training the CNN2$_\text{M}$ model [32]. We evaluate ORCA on P-Lenet$_\text{M}$ in Table IV.

On the CIFAR dataset, which is a harder classification problem than MNIST, the improvements are even more pronounced. While training CNN3$_\text{C}$ [32], ORCA outperforms PIRANHA's training of P-VGG16$_\text{C}$ by 4% in accuracy, by 19× in latency, and by 123× in communication. We achieve the reported accuracy of 59% by running just 2 training epochs. When training for 5 epochs, we achieve a much higher accuracy of 70% (15% better than PIRANHA) while still being 7× faster and requiring 49× lower communication than PIRANHA. We evaluate ORCA on P-VGG16$_\text{C}$ in Table IV.

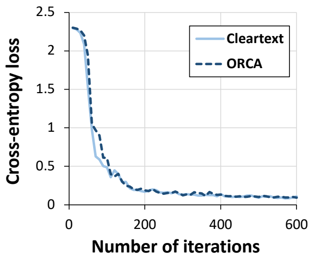
Accuracy: Since ORCA evaluates the models by Gupta et al. [32] faithfully, there are no accuracy gaps between cleartext training and secure training. We show this empirically as well in Figure 5. The minor deviations between the two are because of the randomness introduced by stochastic truncations. Training these models with PIRANHA approximations leads to 10% accuracy.

B. *Comparison with baselines*

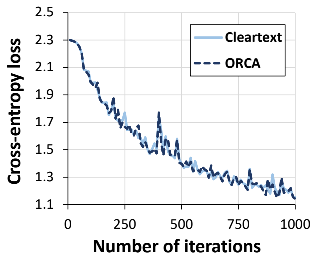
Next, we compare ORCA and baselines on identical models. In particular, we compare ORCA with GPU-based PIRANHA on the models that PIRANHA supports in Section VIII-B1. In Section VIII-B2, we compare ORCA and state-of-the-art CPU-based secure training [39].

1) GPU baselines: We compare ORCA and PIRANHA in Table IV on the benchmarks used by PIRANHA for both training and inference tasks. We observe that ORCA is up to

[3]Since the key generation in ORCA is GPU-optimized, almost all of the offline time of ORCA is spent in moving keys (of sizes given in Table VII) from the machine running the dealer to the machines holding secret data.



(a) CNN2 on MNIST (trained for 1 epoch with batch size 100).



(b) CNN3 on CIFAR-10 (trained for 2 epochs with batch size 100).

Fig. 5: Cross-entropy loss over the test set as a function of the number of training iterations.

		Time (ms)		Comm. (MB)	
Model	Task	PIRANHA	ORCA	PIRANHA	ORCA
P-SecureML$_\text{M}$	training	166 (2.3×)	72	31 (5.68×)	5.45
	inference	57 (9.5×)	6	21 (8.2×)	2.56
P-LeNet$_\text{M}$	training	720 (4×)	180	474 (7.52×)	63
	inference	402 (8.37×)	48	335 (8.81×)	38
P-AlexNet$_\text{C}$	training	984 (4.94×)	199	606 (5.56×)	109
	inference	424 (7.06×)	60	324 (8.9×)	36
P-VGG16$_\text{C}$	training	18096 (2.81×)	6441	17083 (10.05×)	1700
	inference	14106 (9.39×)	1503	13589 (14.91×)	911

Table IV: Comparison against GPU baseline PIRANHA for one training iteration and inference with a batch size of 128.

9× better in latency and 14× better in communication. In this evaluation, both ORCA and PIRANHA use average pools, linear approximations in softmax, and local truncations. Protocols in ORCA for these approximations are provided in Appendix K.

Since the communication of ORCA is an order of magnitude lower than PIRANHA, in settings with lower bandwidth (e.g., WAN settings), these improvements will be even higher. Using newer PCIe4-enabled SSDs will also further improve ORCA's runtime. For example, one training iteration of P-VGG16$_\text{C}$ requires ORCA to transfer a key of 17.2GB in size from SSD to RAM that takes 6.4s in time on our PCIe3-based SSD (Table IX, Appendix A). Once the keys are in RAM then ORCA takes only 2.4s for the rest of the computation (inclusive of the communication time). If we were to use latest generation PCIe4 enabled SSD, such as Samsung 980 PRO [15], which has double the read bandwidth of our setup's SSD, the time to transfer the key will reduce to 3.2s. Consequently, the latency improvement[4] over PIRANHA will rise from 2.81× to 5.65×.

2) CPU baselines: The latest work in CPU-based secure training is by Keller and Sun [39], shown as KS in Table V. Secure training in KS uses faithful maxpools and stochastic

[4]ORCA hides the 2.4s of the current iteration by overlapping it with the 3.2s key read of the next iteration (② of Section III-B).

	Time (in sec)		Comm. (in MB)	
Model	KS	ORCA	KS	ORCA
CNN2$_\text{M}$	10 (7.12×)	1.404	3102 (42×)	73
Model-B$_\text{M}$	17.2 (11.64×)	1.478	5796 (58×)	100
AlexNet$_\text{C}$	299.3 (111.06×)	2.695	46213 (155×)	299
CNN3$_\text{C}$	1685 (441.21×)	3.819	177152 (325×)	545

Table V: Comparison against CPU baseline KS [39] for one training iteration with batch size of 100.

Dataset	Model	Task	LLAMA	Our CPU	Our GPU
MNIST	CNN2$_\text{M}$	training	3.3 (2.35×)	2.8 (2×)	1.404
		inference	1.5 (37.5×)	1.0 (25×)	0.040
CIFAR-10	CNN3$_\text{C}$	training	33.4 (8.75×)	22.8 (5.97×)	3.819
		inference	28.2 (42.15×)	18.6 (27.8×)	0.669
ImageNet	VGG16$_I$	inference	688 (53.7×)	370 (29×)	12.8

Table VI: Comparison of latency (in seconds) of a single iteration of training and inference between LLAMA, our CPU and GPU implementations. For ImageNet, batch size is 16 and the others use batch size of 100.

truncations, and does not support local truncations. To measure running time of KS, we instrument MP-SPDZ [37] to measure the time taken by one iteration of online phase after the preprocessing material has been loaded in RAM. Table V shows that ORCA (with stochastic truncations, maxpools and floating-point softmax) is up to $441\times$ faster than KS, while incurring up to $325\times$ less communication.

C. Improvements breakup

LLAMA [31] is the state-of-the-art in FSS-based protocols for ML tasks, and outperforms other tools such as AriaNN [55]. Conceptually, ORCA makes two performance improvements over LLAMA. It provides new protocols for operations occurring in ML tasks with smaller FSS keys and accelerates these protocols with GPUs. In Table VI, we investigate the individual contributions of protocols and GPUs to the overall speedups over LLAMA. Note that ORCA incurs slightly lower communication than LLAMA (Appendix A). Hence, we only compare time in Table VI.

The protocols of ORCA, described in Section V show up to $5\times$ reduction in key size over LLAMA and $< 2\times$ reduction in number of AES calls (see Table VII). As key read is not the performance bottleneck in CPU implementation, new protocols translate to $< 2\times$ improvement in running time of ORCA run on CPU over LLAMA in Table VI. GPU acceleration of our protocols brings running time further down by $2 - 29\times$. Reading huge FSS keys becomes a performance bottleneck after accelerating compute with GPUs and key size reduction through our new protocols becomes crucial. It is easy to see that without our new protocols, runtime of ORCA with GPUs would worsen by up to $5\times$, that is exactly by the factor of improvement in key size (calculated based on read bandwidth from SSD to RAM).

Dataset	Model	Task	LLAMA	ORCA
MNIST	CNN2$_\text{M}$	training	1.97 (3.34×)	0.59
		inference	1.45 (3.82×)	0.38
CIFAR-10	CNN3$_\text{C}$	training	32.90 (3.20×)	10.28
		inference	28.46 (3.38×)	8.43
ImageNet	VGG16$_I$	inference	714 (4.86×)	147.2

Table VII: Comparison of key size (in GB) against LLAMA for a single iteration of training and inference. For ImageNet, batch size is 16 and the others use batch size of 100.

The inference tasks show better acceleration than training tasks because of floating-point softmax. Recall that softmax is only present in training, not in inference, and softmax's exponentiations use a floating-point 2PC protocol from prior work [51] that hasn't been accelerated with either FSS or GPUs. We find that once the rest of the compute has been accelerated with GPUs, softmax accounts for a considerable portion of the training runtime. For example, one training iteration of CNN2$_\text{M}$ takes about 1.4s once keys are in RAM. Of this, softmax takes $\approx$1.3s, which is $\approx 92\%$ of training time.

We show that GPU accelerates both linear and non-linear layers by an order of magnitude in Figure 6, Appendix A.

ImageNet inference. The last row of Table VI shows that ORCA enables sub-second ImageNet-scale inference, which is an order of magnitude better than what is reported by the prior secure inference works [31], [33], [58] over models with fewer parameters like ResNet50/ResNet152. The Top-1 accuracy of secure inference is 71% and matches the floating-point accuracy over a validation set of 50,000 images.

IX. Related Work

Secure training is a rich area with various techniques. There are solutions that lack cryptographic security guarantees like Trusted Execution Environments [67] and Federated Learning [56]. There are complementary techniques like Differential Privacy (DP) [16], [61] that dictate what training algorithms preserve privacy and ORCA can run such algorithms securely. Among the cryptographic techniques for secure multiparty training, various threat models have been explored:

Two party training. Two parties holding secret data train a joint model with 2PC protocols [17], [36], [47].

Two party training with dealer. A trusted dealer provides correlated randomness to two parties that can then run their 2PC protocols more efficiently. FSS-based protocols fall in this category [55], [65].

Three party honest majority. Three non-colluding parties run a secure training protocol [41], [46], [49], [58], [60], [62].

M-party dishonest majority. M parties, each holding secret data, train a joint model while honest parties are protected from any number of dishonest parties [25], [39], [40], [68].

Works that accelerate non-FSS protocols with GPUs include [29], [34], [45], [48], [50], [58], [64]. Prior works in secure inference that don't address training include [23], [24], [31], [33], [35], [42], [43], [45], [52]–[54]. Similar to all prior works on secure training of DNNs, we have limited ourselves to semi-honest adversaries as malicious security entails additional performance overheads [26], [28], [37], [41], [49], [68].

X. Conclusion

ORCA takes a step towards practical secure inference and training by accelerating FSS-based protocols through both system advances with GPUs and new cryptographic techniques to reduce the size of FSS keys. Together, the time to securely train CIFAR models has reduced to an hour and ImageNet-scale inference runs in sub-second. We also identify concrete challenges for future work: integrating newer hardware (PCIe5 comes out this year) and GPU-accelerated FSS-based protocols for accurate softmax.

References

[1] "Basic Linear Algebra on NVIDIA GPUs," https://developer.nvidia.com/cublas.

[2] "CUDA," https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/contents.html.

[3] "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[4] "cuRAND," https://developer.nvidia.com/curand.

[5] "CUTLASS," https://github.com/NVIDIA/cutlass.

[6] "Enhancing Memory Allocation with New NVIDIA CUDA 11.2 Features," https://developer.nvidia.com/blog/enhancing-memory-allocation-with-new-cuda-11-2-features/.

[7] "NVIDIA cuDNN," https://developer.nvidia.com/cudnn.

[8] "NVIDIA Nsight Compute," https://developer.nvidia.com/nsight-compute.

[9] "Page-Locked Host Memory," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#page-locked-host-memory.

[10] "PCI Express," https://en.wikipedia.org/wiki/PCI_Express.

[11] "PyTorch/CSPRNG," https://github.com/pytorch/csprng.

[12] "SecFloat: Accurate floating-point meets secure 2-party computation," https://github.com/mpc-msri/EzPC.

[13] "Using CUDA Warp-Level Primitives," https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[14] "VGG16 for ImageNet," https://github.com/minar09/VGG16-PyTorch.

[15] "Samsung 980 PRO SSD," https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-2tb-mz-v8p2t0b-am/, 2016.

[16] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep learning with differential privacy," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[17] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: two-party secure neural network training and prediction," in *CCS*, 2019.

[18] E. Boyle, N. Chandran, N. Gilboa, D. Gupta, Y. Ishai, N. Kumar, and M. Rathee, "Function secret sharing for mixed-mode and fixed-point secure computation," in *EUROCRYPT*, 2020.

[19] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing," in *EUROCRYPT*, 2015.

[20] ——, "Function secret sharing: Improvements and extensions," in *CCS*, 2016.

[21] ——, "Secure computation with preprocessing via function secret sharing," in *TCC*, 2019.

[22] R. Canetti, "Security and Composition of Multiparty Cryptographic Protocols," *J. Cryptology*, 2000.

[23] N. Chandran, D. Gupta, S. L. B. Obbattu, and A. Shah, "Simc: Ml inference secure against malicious clients at semi-honest cost," in *USENIX Security Symposium*, 2022.

[24] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "Astra: High throughput 3pc over rings with application to secure prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW'19, 2019.

[25] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh, "Maliciously secure matrix multiplication with applications to private deep learning," in *Advances in Cryptology – ASIACRYPT 2020*, S. Moriai and H. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 31–59.

[26] A. P. K. Dalskov, D. Escudero, and M. Keller, "Secure evaluation of quantized neural networks," *PoPETs*, 2020.

[27] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[28] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *CRYPTO*, 2020.

[29] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen, "Faster maliciously secure two-party computation using the gpu," in *Security and Cryptography for Networks*, M. Abdalla and R. De Prisco, Eds. Cham: Springer International Publishing, 2014, pp. 358–379.

[30] O. Goldreich, S. Micali, and A. Wigderson, "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority," in *STOC*, 1987.

[31] K. Gupta, D. Kumaraswamy, N. Chandran, and D. Gupta, "Llama: A low latency math library for secure inference," in *PETS*, 2022.

[32] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision." in *ICML*, vol. 37, 2015, pp. 1737–1746.

[33] Z. Huang, W. jie Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure two-party deep neural network inference," in *USENIX Security Symposium*, 2022.

[34] N. Husted, S. Myers, A. Shelat, and P. Grubbs, "Gpu and cpu parallelization of honest-but-curious secure two-party computation," in *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[35] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "Gazelle: A low latency framework for secure neural network inference," in *USENIX Security Symposium*, 2018.

[36] M. Kelkar, P. H. Le, M. Raykova, and K. Seth, "Secure poisson regression," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 791–808.

[37] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation," in *CCS*, 2020.

[38] M. Keller and K. Sun, "Effectiveness of mpc-friendly softmax replacement," *arXiv preprint arXiv:2011.11202*, 2020.

[39] ——, "Secure quantized training for deep learning," in *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, 2022.

[40] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure multi-party computation meets machine learning," in *NeurIPS*, 2021.

[41] N. Koti, M. Pancholi, A. Patra, and A. Suresh, "SWIFT: super-fast and robust privacy-preserving machine learning," in *USENIX Security Symposium*, 2021.

[42] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *IEEE S&P*, 2020.

[43] R. Lehmkuhl, P. Mishra, A. Srinivasan, and R. A. Popa, "Muse: Secure inference resilient to malicious clients," in *USENIX Security Symposium*, 2021.

[44] Y. Lindell, "How to simulate it – a tutorial on the simulation proof technique," *Tutorials on the Foundations of Cryptography*, 2017.

[45] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *USENIX Security Symposium*, 2020.

[46] P. Mohassel and P. Rindal, "ABY^3: A Mixed Protocol Framework for Machine Learning," in *CCS*, 2018.

[47] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *IEEE S&P*, 2017.

[48] L. K. L. Ng and S. S. M. Chow, "Gforce: Gpu-friendly oblivious and rapid neural network inference," in *USENIX Security Symposium*, 2021.

[49] A. Patra and A. Suresh, "Blaze: Blazing fast privacy-preserving machine learning," in *NDSS*, 2020.

[50] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa, "Visor: Privacy-preserving video analytics as a cloud service," in *USENIX Security Symposium*, 2020.

[51] D. Rathee, A. Bhattacharya, R. Sharma, D. Gupta, N. Chandran, and A. Rastogi, "SecFloat: Accurate Floating-Point meets Secure 2-Party Computation," in *IEEE S&P*, 2022.

[52] D. Rathee, M. Rathee, R. K. K. Goli, D. Gupta, R. Sharma, N. Chandran, and A. Rastogi, "SIRNN: A math library for secure inference of RNNs," in *IEEE S&P*, 2021.

[53] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow2: Practical 2-Party Secure Inference," in *CCS*, 2020.

[54] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *ASIACCS*, 2018.

[55] T. Ryffel, D. Pointcheval, and F. Bach, "ARIANN: Low-interaction privacy-preserving deep learning via function secret sharing," in *PETS*, 2022.

[56] S. Sav, A. Pyrgelis, J. R. Troncoso-Pastoriza, D. Froelicher, J. Bossuat, J. S. Sousa, and J. Hubaux, "POSEIDON: privacy-preserving federated neural network learning," in *NDSS*, 2021.

[57] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *ICLR*, 2015.

[58] S. Tan, B. Knott, Y. Tian, and D. J. Wu, "Cryptgpu: Fast privacy-preserving machine learning on the GPU," in *IEEE S&P*, 2021.

[59] C. Tezcan, "Optimization of advanced encryption standard on graphics processing units," *IEEE Access*, vol. 9, pp. 67 315–67 326, 2021.

[60] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," *PoPETs*, 2019.

[61] S. Wagh, X. He, A. Machanavajjhala, and P. Mittal, "Dp-cryptography: Marrying differential privacy and cryptography in emerging applications," *Commun. ACM*, 2021.

[62] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "Falcon: Honest-majority maliciously secure framework for private deep learning," *PoPETs*, 2021.

[63] K. Wang, D. Fussell, and C. Lin, "Fast Fine-Grained Global Synchronization on GPUs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 793–806.

[64] J.-L. Watson, S. Wagh, and R. A. Popa, "Piranha: A GPU Platform for Secure Computation," in *USENIX Security Symposium*, 2022.

[65] P. Yang, Z. L. Jiang, S. Gao, J. Zhuang, H. Wang, J. Fang, S. Yiu, and Y. Wu, "Fssnn: Communication-efficient secure neural network training via function secret sharing," Cryptology ePrint Archive, Paper 2023/073, 2023, https://eprint.iacr.org/2023/073. [Online]. Available: https://eprint.iacr.org/2023/073

[66] A. C. Yao, "Protocols for secure computations," in *FOCS*, 1982.

[67] P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana, "Plinius: Secure and persistent machine learning model training," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[68] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Helen: Maliciously secure coopetitive learning for linear models," in *IEEE S&P*, 2019.

APPENDIX

A. Additional Empirical Results

1) Microbenchmarks: We show acceleration of GPUs for linear layers and non-linear layers separately in Figure 6. Across, microbenchmarks of varying sizes, the GPU-based protocols in ORCA are an order of magnitude more efficient compared to their CPU counterparts.

2) Online communication: We show that our new protocols in ORCA not only reduce key size and online compute but also online communication. Concrete numbers are provided in Table VIII.

Dataset	Model	Task	LLAMA	ORCA
MNIST	CNN2$_\mathsf{M}$	training	76 (1.04×)	73
		inference	22 (1.19×)	19
CIFAR-10	CNN3$_\mathsf{C}$	training	579 (1.06×)	545
		inference	365 (1.19×)	307

Table VIII: Comparison of communication (in MB) of a single iteration of training and inference between LLAMA and ORCA. Batch size of 100 is being used.

Model	Key size (in GB)	Key read (ms)	Compute (ms)	ORCA (ms)
P-SecureML$_\mathsf{M}$	0.04	12	**72**	72
P-LeNet$_\mathsf{M}$	0.41	169	**180**	180
CNN2$_\mathsf{M}$	0.59	251	**1404**	1404
Model-B$_\mathsf{M}$	0.98	413	**1478**	1478
P-AlexNet$_\mathsf{C}$	0.38	172	**199**	199
AlexNet$_\mathsf{C}$	7.17	**2695**	1921	2695
CNN3$_\mathsf{C}$	10.28	**3819**	2247	3819
P-VGG16$_\mathsf{C}$	17.2	**6441**	2422	6441

Table IX: Key size, time of key read and compute for ORCA for different models. The last column reports the larger of key read time and compute time.

3) Split of ORCA *time between key read and compute:* Table IX shows the time ORCA spends on reading keys and on compute (including time spent on communication) for one training iteration. Since key read and compute overlap in ORCA, a training iteration takes the larger of the two times. As the models become bigger, key read starts to dominate.

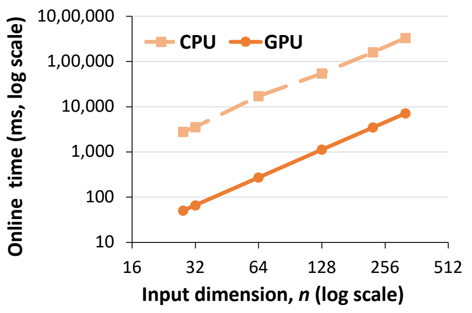
B. Formal Correctness and Security of FSS Scheme

Definition 4 (FSS: Correctness and Security [19], [20])**.** *Let $\mathcal{G} = \{g\}$ be a function family, $P_\mathcal{G} = \{\hat{g}\}$ be the set of descriptions of functions in $\mathcal{G}$, and* Leak *be a function specifying the allowable leakage about $\hat{g}$. When* Leak *is omitted, it is understood to output only $\mathbb{G}^{\mathsf{in}}$ and $\mathbb{G}^{\mathsf{out}}$. We say that* $(\mathsf{Gen}, \mathsf{Eval})$ *as in Definition 1 is an FSS scheme for $\mathcal{G}$ (with respect to leakage* Leak*) if it satisfies the following.*

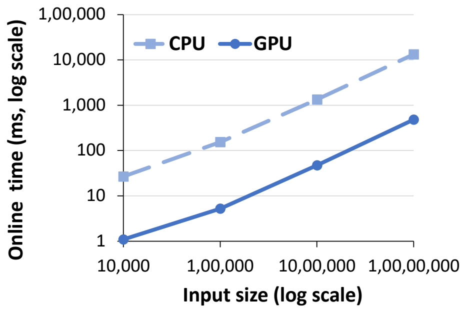
- **Correctness:** *For all $\hat{g} \in P_\mathcal{G}$ describing $g : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$, and every $x \in \mathbb{G}^{\mathsf{in}}$, if $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{g})$ then* $\Pr[\mathsf{Eval}(0, k_0, x) + \mathsf{Eval}(1, k_1, x) = g(x)] = 1$.
- **Security:** *For each $b \in \{0, 1\}$ there is a PPT algorithm Sim_b (simulator), such that for every sequence $(\hat{g}_\lambda)_{\lambda \in \mathbb{N}}$ of polynomial-size function descriptions from $\mathcal{G}$ and polynomial-size input sequence x_λ for g_λ, the outputs of the following* Real *and* Ideal *experiments are computationally indistinguishable:*
  - Real_λ*:* $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{g}_\lambda)$*; Output k_b.*
  - Ideal_λ*: Output* $\mathsf{Sim}_b(1^\lambda, \mathsf{Leak}(\hat{g}_\lambda))$.

C. Our Threat Model

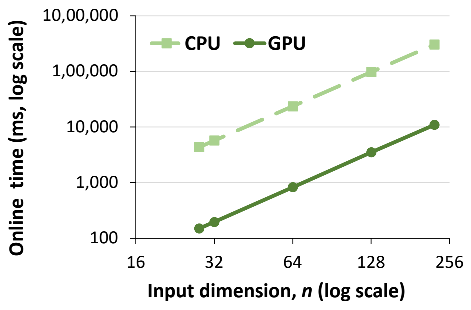
2-party secure computation (2PC) enables two parties P_0 and P_1, with private inputs x_0 and x_1 respectively, to compute any public joint function $y = f(x_0, x_1)$ on their inputs. Informally, security requires that they only learn y and nothing



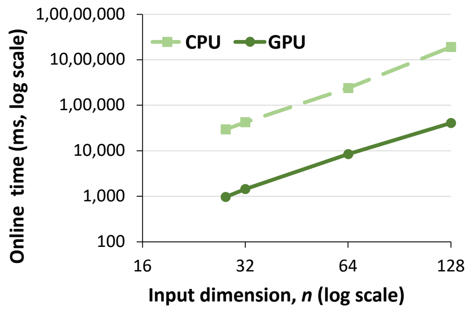
(a) Convolution on an $n \times n \times 64$ input with a 5×5 kernel, 64 output channels, stride 1, and padding 1. $n = 28,\ 32,\ 64,\ 128,\ 224,\ 320$.



(b) Stochastic-Truncation + ReLU on inputs of increasing sizes. Input size $= 10^4,\ 10^5,\ 10^6,\ 10^7$.



(c) Stochastic-Truncation + ReLU + MaxPool on an $n \times n \times 64$ input with a 3×3 kernel and stride 1. $n = 28,\ 32,\ 64,\ 128,\ 224$.



(d) Stochastic-Truncation + ReLU + MaxPool on an $n \times n \times 64$ input with a 11×11 kernel and stride 1. $n = 28,\ 32,\ 64,\ 128$.

Fig. 6: Comparison of online time on CPU and GPU

else through the course of an interactive protocol which they execute. We consider 2PC in the trusted dealer model. That is, there exists a trusted dealer that provides correlated randomness to the two parties in a pre-processing phase (before inputs to the computation are available). Security is proven in the simulation paradigm [22], [30], [44] against a semi-honest static probabilistic polynomial time (PPT) adversary that corrupts one of the two parties. That is, the adversary corrupts either P_0 or P_1 before the protocol begins. The corrupted party is guaranteed to follow the protocol specification but may try to learn additional information from the protocol. Security is modelled by defining two worlds: a *real* world in which P_0 and P_1 interact with each other through the protocol in the presence of adversary $\mathcal{A}$ and the environment $\mathcal{Z}$; and an *ideal* world in which the parties send their inputs to a trusted functionality computing $f(x_0, x_1)$ faithfully. Security requires that for every real-world adversary, there exists an ideal world adversary (called the simulator $\mathcal{S}$) such that no environment $\mathcal{Z}$ can distinguish between the two worlds.

D. Proof of Lemma 1

Lemma 1. *Consider a masked value $\hat{x} \in \mathbb{U}_N$ with underlying value x and random mask $r^{(x)}$. Let $\hat{y} = \mathsf{TR}(\hat{x}, f)$ and $r^{(y)} = \mathsf{TR}(r^{(x)}, f)$. Then the following holds:*

$$x \gg_{\mathsf{st}} f = \mathsf{SignExt}_{n-f,n}(\hat{y} - r^{(y)})$$

Proof. To prove the above lemma, it suffices to show that $\hat{y} - r^{(y)} \mod 2^{(n-f)}$ is equal to $x \gg_{\mathsf{st}} f$ in $(n-f)$ bits as it is then sign-extended to n bits. We begin the proof by stating the fact that when output bitwidth is $(n-f)$, the output of logical right-shift is the same as arithmetic right-shift, as the two only differ in the most significant f bits, which are removed when the output is reduced to $(n-f)$ bits.

So, for a given masked input $\hat{x} \in \mathbb{U}_N$, with underlying value $x \in \mathbb{U}_N$ and random mask $r \in \mathbb{U}_N$, it suffices to prove that:

$$\mathsf{TR}(\hat{x}, f) - \mathsf{TR}(r, f) = \begin{cases} \mathsf{TR}(x, f) & \text{with prob. } 1 - t \\ \mathsf{TR}(x, f) + 1 & \text{with prob. } t \end{cases}$$

where $t = (x \mod 2^f) \cdot 2^{-f}$. To show this, consider:

$$\begin{aligned} \mathsf{TR}(x, f) &= \mathsf{TR}(\hat{x} - r \mod 2^n, f) \\ &= \mathsf{TR}(\hat{x} - r + 2^n \cdot 1\{\hat{x} < r\}, f) \\ &= (\hat{x} - r + 2^n \cdot 1\{\hat{x} < r\}) \gg_L f \mod 2^{(n-f)} \\ &= (\hat{x} - r) \gg_L f + 2^{n-f} \cdot 1\{\hat{x} < r\} \mod 2^{(n-f)} \\ &= (\hat{x} - r) \gg_L f \end{aligned}$$

Let $\hat{x}_1, r_1 \in \mathbb{U}_{2^{n-f}}$ and $\hat{x}_0, r_0 \in \mathbb{U}_{2^f}$ be numbers such that, $\hat{x} = \hat{x}_1 \cdot 2^f + \hat{x}_0$, $r = r_1 \cdot 2^f + r_0$. So, $\hat{x}_1 = \mathsf{TR}(\hat{x}, f)$ and

$r_1 = \mathsf{TR}(r, f)$.

$$\begin{aligned}\mathsf{TR}(x, f) &= (\hat{x} - r) \gg_L f \\ &= \hat{x}_1 - r_1 + (\hat{x}_0 - r_0) \gg_L f \\ &= \hat{x}_1 - r_1 + (\hat{x}_0 - r_0) \gg_A f\end{aligned}$$

Note that the term $(\hat{x}_0 - r_0) \gg_A f$ can only take values 0 or -1 as $-2^f < \hat{x}_0 - r_0 < 2^f$. So, we have:

$$\begin{aligned}\mathsf{TR}(x, f) &= \hat{x}_1 - r_1 - 1\{\hat{x}_0 < r_0\} \\ \implies \hat{x}_1 - r_1 &= \mathsf{TR}(x, f) + 1\{\hat{x}_0 < r_0\} \\ \mathsf{TR}(\hat{x}, f) - \mathsf{TR}(r, f) &= \mathsf{TR}(x, f) + 1\{\hat{x}_0 < r_0\}\end{aligned}$$

Now, to complete the proof, we only need to show that the term $1\{\hat{x}_0 < r_0\}$ takes values 0 with probability $1 - t$. We have:

$$\begin{aligned}\hat{x} &= x + r \mod 2^n \\ \implies \hat{x}_0 &= (x \mod 2^f) + r_0 \mod 2^f\end{aligned}$$

r_0 is a uniformly random value. Consider the case when $r_0 < 2^f - (x \mod 2^f)$. In this case,

$$\begin{aligned}& r_0 + (x \mod 2^f) < 2^f \\ \implies & \hat{x}_0 = (x \mod 2^f) + r_0 \mod 2^f \\ & \quad = (x \mod 2^f) + r_0 \\ \implies & \hat{x}_0 \geqslant r_0 \\ \implies & 1\{\hat{x}_0 < r_0\} = 0\end{aligned}$$

For this to happen, r_0 needs to randomly assume the value from the set $\{0, 1, \dots 2^f - (x \mod 2^f) - 1\}$ which happens with probability $(2^f - (x \mod 2^f))/2^f = 1 - t$. □

E. Proof for the correctness of DReLU offset function

While there exists an expression for the offset function of DReLU in the previous work of [18], we derive an alternate expression that benefits from the fact that all the private comparisons done in it are by the corresponding input mask only and hence, the DCF key required to do this can be re-used in other protocols like Sign-Extension.

Lemma 2. *Offset function for* DReLU_n, *defined as* $\mathsf{DReLU}_n(x) = 1\{x < 2^{n-1}\}$ *for* $x \in \mathbb{U}_N$ *is:*

$$\begin{aligned}\mathsf{DReLU}_n^{[\mathsf{r}^{in}, \mathsf{r}^{out}]}(\hat{x}) &= 1\{\hat{y} < \mathsf{r}^{in}\} - 1\{\hat{x} < \mathsf{r}^{in}\} \\ &\quad + 1\{\hat{y} \geqslant 2^{n-1}\} + \mathsf{r}^{out}\end{aligned}$$

where $\hat{y} = \hat{x} + 2^{n-1} \mod 2^n$.

Proof. Starting with the definition of offset function, we get:

$$\begin{aligned}{\mathsf{DReLU}_n}^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(\hat{x}) &= \mathsf{DReLU}_n(\hat{x} - \mathsf{r}^{\mathsf{in}} \mod 2^n) + \mathsf{r}^{\mathsf{out}} \\ &= 1\{\hat{x} - \mathsf{r}^{\mathsf{in}} \mod 2^n < 2^{n-1}\} + \mathsf{r}^{\mathsf{out}}\end{aligned}$$

Let $\hat{y} = \hat{x} + 2^{n-1} \mod 2^n$ and $r' = \mathsf{r}^{\mathsf{in}} + 2^{n-1} \mod 2^n$. According to the above equation, the indicator function above returns 1 when $\hat{x}$ lies in the region starting from r^{in} until r', taking care of wraps. Now, we will simplify this expression by considering two cases.

Case 1: $\mathsf{r}^{\mathsf{in}} < 2^{n-1}$: This implies that $\mathsf{r}^{\mathsf{in}} + 2^{n-1} < 2^n$ and $r' = \mathsf{r}^{\mathsf{in}} + 2^{n-1}$. Hence, the expression becomes:

$$\begin{aligned}\mathsf{DReLU}_n^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(\hat{x}) &= 1\{\hat{x} \in [\mathsf{r}^{\mathsf{in}}, r')\} + \mathsf{r}^{\mathsf{out}} \\ &= 1\{\hat{x} < r'\} - 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}}\end{aligned}$$

Case 2: $\mathsf{r}^{\mathsf{in}} \geqslant 2^{n-1}$: This implies that $\mathsf{r}^{\mathsf{in}} + 2^{n-1} \geqslant 2^n$ and $r' = \mathsf{r}^{\mathsf{in}} - 2^{n-1}$. Hence, the expression becomes:

$$\begin{aligned}\mathsf{DReLU}_n^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(\hat{x}) &= 1\{\hat{x} \in [0, r') \cup [\mathsf{r}^{\mathsf{in}}, 2^n - 1]\} + \mathsf{r}^{\mathsf{out}} \\ &= 1\{\hat{x} < r'\} + 1\{\hat{x} \geqslant \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}} \\ &= 1\{\hat{x} < r'\} + 1 - 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}}\end{aligned}$$

Since the expressions in the two cases differ exactly by 1, combining the two cases, we get:

$$\begin{aligned}\mathsf{DReLU}_n^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(x) &= 1\{\mathsf{r}^{\mathsf{in}} \geqslant 2^{n-1}\} + 1\{\hat{x} < r'\} \\ &\quad - 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}} \qquad (2)\end{aligned}$$

Now, to further simplify the $1\{\hat{x} < r'\}$ term, we use the following lemma from [18]:

Lemma 3 (Lemma 1 from [18])**.** *Let* $a, \tilde{a}, b, \tilde{b}, r \in \mathbb{U}_N$, *where* $a \leq b$, $\tilde{a} = a + r \bmod N$ *and* $\tilde{b} = b + r \bmod N$. *Define 4 boolean predicates over* $\mathbb{U}_N \to \{0, 1\}$ *as follows:* $P(x)$ *denotes* $x < \tilde{a}$, $P'(x)$ *denotes* $x \leq \tilde{a}$, $Q(x)$ *denotes* $(x + (b - a) \bmod N) < \tilde{b}$, $Q'(x)$ *denotes* $(x + (b - a) \bmod N) \leq \tilde{b}$. *Then, the following holds:*

$$P(x) = Q(x) + (e_a - e_x) \textit{ and } P'(x) = Q'(x) + (e_a - e_x)$$

$$\textit{where } e_a = 1\{\tilde{a} + (b - a) > N - 1\} \textit{ and } e_x = 1\{x + (b - a) > N - 1\}$$

In the above lemma, if we set $a = 0, b = 2^{n-1}, r = \mathsf{r}^{\mathsf{in}}, x = \hat{y}$, we get:

$$\begin{aligned}\tilde{a} &= a + r \mod N \\ &= \mathsf{r}^{\mathsf{in}} \\ \tilde{b} &= b + r \mod N \\ &= \mathsf{r}^{\mathsf{in}} + 2^{n-1} \mod N \\ &= r' \\ P(\hat{y}) &= 1\{\hat{y} < \tilde{a}\} \\ &= 1\{\hat{y} < \mathsf{r}^{\mathsf{in}}\} \\ Q(\hat{y}) &= 1\{\hat{y} + (b - a) \bmod N < \tilde{b}\} \\ &= 1\{\hat{y} + 2^{n-1} \bmod N < r'\} \\ &= 1\{\hat{x} < r'\} \\ e_a &= 1\{\tilde{a} + (b - a) > N - 1\} \\ &= 1\{\mathsf{r}^{\mathsf{in}} + 2^{n-1} > 2^n - 1\} \\ &= 1\{\mathsf{r}^{\mathsf{in}} > 2^{n-1} - 1\} \\ &= 1\{\mathsf{r}^{\mathsf{in}} \geqslant 2^{n-1}\} \\ e_{\hat{y}} &= 1\{\hat{y} + (b - a) > N - 1\} \\ &= 1\{\hat{y} + 2^{n-1} > 2^n - 1\}\end{aligned}$$

$$= 1\{\hat{y} > 2^{n-1} - 1\}$$
$$= 1\{\hat{y} \geqslant 2^{n-1}\}$$

We observe that $Q(\hat{y})$ is the term we need. So:

$$Q(\hat{y}) = P(\hat{y}) - e_a + e_{\hat{y}}$$
$$1\{\hat{x} < r'\} = 1\{\hat{y} < \mathsf{r}^{\mathsf{in}}\} - 1\{\mathsf{r}^{\mathsf{in}} \geqslant 2^{n-1}\} + 1\{\hat{y} \geqslant 2^{n-1}\}$$

Plugging this result in Equation 2, we get:

$$\begin{aligned}\mathsf{DReLU}_n{}^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}(\hat{x}) &= 1\{\hat{x} < r'\} + 1\{\mathsf{r}^{\mathsf{in}} \geqslant 2^{n-1}\} - 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}} \\ &= 1\{\hat{y} < \mathsf{r}^{\mathsf{in}}\} - 1\{\mathsf{r}^{\mathsf{in}} \geqslant 2^{n-1}\} + 1\{\hat{y} \geqslant 2^{n-1}\} + 1\{\mathsf{r}^{\mathsf{in}} \geqslant 2^{n-1}\} - 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}} \\ &= 1\{\hat{y} < \mathsf{r}^{\mathsf{in}}\} - 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} + 1\{\hat{y} \geqslant 2^{n-1}\} + \mathsf{r}^{\mathsf{out}}\end{aligned}$$

□

F. Stochastic-Truncation + ReLU + MaxPool

In many networks, it is common to have a convolution layer, a ReLU layer, and a MaxPool layer in succession. Since convolution is always followed by truncation in the case of fixed-point training, truncation, ReLU, and MaxPool occur in succession. Hence, we need a protocol for *stochastic-truncation + ReLU + MaxPool*. We note that the result of this fused operation doesn't depend on the order in which they are applied. As a result, many existing works carry out ReLU after MaxPool [42], [53], as this reduces the number of elements on which ReLU is calculated.

LLAMA [31] provides a protocol for MaxPool, which internally uses the spline-based protocol for ReLU from [18] to calculate max of two elements. First, we replace this ReLU with our protocol Π^{ReLU} and follow a three-step approach to implement a protocol for stochastic-truncation + ReLU + MaxPool with bitwidth n and precision f:

1) Locally truncate-reduce the input to $(n-f)$-bits.
2) Apply the modified protocol for uniform bitwidth MaxPool to the $(n-f)$-bit truncated output.
3) Apply the protocol for $\mathsf{ReLUExt}_{n-f,n}$ to the resulting $(n-f)$-bit values to get result in n-bits.

In the above protocol, all the comparisons are done over $(n-f)$-bits instead of n bits and use our key optimized ReLU protocol, giving us significant savings over a protocol designed using building blocks in [31]. The following theorem summarizes the cost when the MaxPool is done over k elements.

Theorem 10. *There exists a protocol* $\Pi^{\mathsf{TRM}}_{n,f,k}$ *which realizes stochastic-truncation* + ReLU + MaxPool *of k elements securely such that* $\mathsf{keysize}(\Pi^{\mathsf{TRM}}_{n,f,k}) = (k-1)\cdot\mathsf{keysize}(\hat{\Pi}^{\mathsf{ReLU}}_{n-f}) + \mathsf{keysize}(\Pi^{\mathsf{ReLUExt}}_{n-f,n})$. *Its online phase requires $k-1$ evaluations of $\hat{\Pi}^{\mathsf{ReLU}}_{n-f}$, 1 evaluation of $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$ and communication of $2(k-1)(n-f+1)+8$ bits in $2k-2$ rounds.*

In contrast, the baseline [31] has keysize roughly $k\cdot\mathsf{keysize}(\mathsf{DCF}_{n,2n}+5n)+\mathsf{keysize}(\mathsf{DCF}_{n-1,2n})+3n$. For $n = 64, f = 24$ used in our benchmarks, for a MaxPool of size 3×3, that is, $k=9$, we get a key size reduction of $3.4\times$.

G. Security proof of $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$

For $b \in \{0,1\}$, the simulator $\mathsf{Sim}^{\mathsf{ReLUExt}}_b$ for ReLUExt is given $\hat{x}$ and u_b, and has to simulate the view of party b, i.e., messages $k^{<}_b || r^{(d)}_b || r^{(w)}_b || \boldsymbol{p}_b || \boldsymbol{q}_b$ from the dealer and $\hat{w}_{1-b} || \hat{d}_{1-b}$ from the other evaluator. The simulator follows the following steps:

1) Randomly samples $w_{b,\mathsf{sim}}$ and generates $k^{<}_{b,\mathsf{sim}}$ using $\mathsf{Sim}^{<}_b$ with input $\hat{x}$ and output $w_{b,\mathsf{sim}}$.
2) Randomly samples $r^{(w)}_{b,\mathsf{sim}}, r^{(d)}_{b,\mathsf{sim}}, \hat{w}_{1-b,\mathsf{sim}}, \hat{d}_{1-b,\mathsf{sim}} \in \mathbb{U}_4$.
3) Calculates $\hat{w}_{b,\mathsf{sim}}$ and $\hat{d}_{b,\mathsf{sim}}$ using steps $2, 4$ and 5 from $\mathsf{Eval}^{\mathsf{ReLUExt}}_{n-f,n}$.
4) Sets $\hat{w}_{\mathsf{sim}} = \hat{w}_{b,\mathsf{sim}} + \hat{w}_{1-b,\mathsf{sim}} \mod 4$ and $\hat{d}_{\mathsf{sim}} = \hat{d}_{b,\mathsf{sim}} + \hat{d}_{1-b,\mathsf{sim}} \mod 4$.
5) Sets $\hat{i}_{\mathsf{sim}} = 2\cdot\hat{d}_{\mathsf{sim}} + \hat{w}_{\mathsf{sim}} \mod 4$.
6) Randomly samples $\boldsymbol{p}_{b,\mathsf{sim}} \in \mathbb{U}^4_N$ and sets $\boldsymbol{p'}_{b,\mathsf{sim}} = \boldsymbol{p'}_{b,\mathsf{sim}} \ggg \hat{i}_{\mathsf{sim}}$.
7) Sets $\hat{j}_{\mathsf{sim}} = \hat{d}_{\mathsf{sim}} \mod 2$.
8) Randomly samples $q_{b,\mathsf{sim},1-\hat{j}_{\mathsf{sim}}} \in \mathbb{U}_N$.
9) Sets $q_{b,\mathsf{sim},\hat{j}_{\mathsf{sim}}} = \hat{u}_b - p'_{b,\mathsf{sim},3}\cdot(\hat{x}+2^{n-f}) + p'_{b,\mathsf{sim},2}\cdot\hat{x}$.
10) Sets $\boldsymbol{q}_{b,\mathsf{sim}} = \{q_{b,\mathsf{sim},0}, q_{b,\mathsf{sim},1}\}$

H. Protocol and security proof for $\Pi^{\mathsf{FixToFloat}}_{n,f}$

We describe the protocol for $\mathsf{FixToFloat}_{n,f}$ in Figure 7. We now provide simulation-based proof of its security. For $b \in \{0,1\}$, the simulator $\mathsf{Sim}^{\mathsf{FixToFloat}}_b$ for FixToFloat is given $\hat{x}$ and (z_b, s_b, e_b, m_b) and has to simulate the view of party b, i.e., messages $k^{(\mathsf{MIC})}_b || k^{(\mathsf{select})}_b || r^{(s)}_b || r^{(u)}_b || r^{(y)}_b || c_b$ from dealer and $\hat{s}_{1-b} || \hat{u}_{1-b} || \hat{y}_{1-b}$ from the other evaluator. The simulator follows the following steps:

1) Randomly samples $r^{(s)}_{b,\mathsf{sim}}$ and $\hat{s}_{1-b,\mathsf{sim}}$ from $\mathbb{U}_2$.
2) Randomly samples $r^{(y)}_{b,\mathsf{sim}}, \hat{y}_{1-b,\mathsf{sim}}$ and $\hat{u}_{1-b,\mathsf{sim}}$ from $\mathbb{U}_N$.
3) Sets $\hat{s}_{\mathsf{sim}} = s_b + r^{(s)}_{b,\mathsf{sim}} + \hat{s}_{1-b,\mathsf{sim}}$.
4) Randomly samples $\hat{y}_{b,\mathsf{sim}}$ constrained to the condition that LSB of $\hat{y}_{b,\mathsf{sim}} + b\cdot\hat{x}$ is 0.
5) Invokes $\mathsf{Sim}^{\mathsf{select}}_b$ with input $(1-\hat{s}_{\mathsf{sim}}, \hat{x})$ and output $(\hat{y}_{b,\mathsf{sim}} + b\cdot\hat{x}) \gg_L 1$ to generate $k^{\mathsf{select}}_{b,\mathsf{sim}}$.
6) Sets $\hat{y}_{\mathsf{sim}} = \hat{y}_{b,\mathsf{sim}} + \hat{y}_{1-b,\mathsf{sim}}$
7) Randomly samples $\hat{u}_{\mathsf{sim}}$ from $\mathbb{U}_N$ and frac from $\mathbb{U}_{2^{n-24}}$.
8) Sets $c_{b,\mathsf{sim}} = m_b\cdot 2^{n-24} + \mathsf{frac} - b\cdot\hat{y}_{\mathsf{sim}}\cdot\hat{u}_{\mathsf{sim}} + r^{(y)}_{b,\mathsf{sim}}\cdot\hat{u}_{\mathsf{sim}} + r^{(u)}_{b,\mathsf{sim}}\cdot\hat{y}_{\mathsf{sim}}$.
9) Sets $\hat{u}_{b,\mathsf{sim}} = \hat{u}_{\mathsf{sim}} - \hat{u}_{1-b,\mathsf{sim}}$.
10) Randomly samples $\boldsymbol{t}_{b,\mathsf{sim}}$ from $\mathbb{U}^{2n}_N$ subject to constraints in line 4,5 and 7 in $\mathsf{Eval}^{\mathsf{FixToFloat}}_{n,f}$.
11) Calculates $r^{(u)}_{b,\mathsf{sim}}$ according to line 8 in $\mathsf{Eval}^{\mathsf{FixToFloat}}_{n,f}$.
12) Invokes $\mathsf{Sim}^{\mathsf{MIC}}_b$ with input $\hat{x}$ and output $\boldsymbol{t}_{b,\mathsf{sim}}$ to generate $k^{\mathsf{MIC}}_{b,\mathsf{sim}}$.

```
**FixToFloat** $\Pi^{\mathsf{FixToFloat}}_{n,f}$
$\mathsf{Gen}^{\mathsf{FixToFloat}}_{n,f}(\mathsf{r}^{\mathsf{in}})$ :
  1: Let $\boldsymbol{p} = \boldsymbol{p}^{(n)}$ and $\boldsymbol{q} = \boldsymbol{q}^{(n)}$
  2: $(k_0^{(\mathsf{MIC})}, k_1^{(\mathsf{MIC})}) \leftarrow \mathsf{Gen}^{\mathsf{MIC}}_{n,\boldsymbol{p},\boldsymbol{q}}(\mathsf{r}^{\mathsf{in}}, 0)$
  3: $r^{(s)} \xleftarrow{\$} \mathbb{U}_2$
  4: $r^{(u)} \xleftarrow{\$} \mathbb{U}_N$
  5: $r^{(\mathsf{select})} \xleftarrow{\$} \mathbb{U}_N$
  6: $(k_0^{(\mathsf{select})}, k_1^{(\mathsf{select})}) \leftarrow \mathsf{Gen}^{\mathsf{select}}_n(r^{(s)}, \mathsf{r}^{\mathsf{in}}, r^{(\mathsf{select})})$
  7: $r^{(y)} = 2 \cdot r^{(\mathsf{select})} - \mathsf{r}^{\mathsf{in}}$
  8: $c = r^{(u)} \cdot r^{(y)}$
  9: share $(r^{(s)}, r^{(u)}, r^{(y)}, c)$
  10: $\forall b \in \{0,1\}, k_b = k_b^{(\mathsf{MIC})} || k_b^{(\mathsf{select})} || r_b^{(s)} || r_b^{(u)} || r_b^{(y)} || c_b$

$\mathsf{Eval}^{\mathsf{FixToFloat}}_{n,f}(b, k_b, \hat{x})$ :
  1: Let $\boldsymbol{p} = \boldsymbol{p}^{(n)}$ and $\boldsymbol{q} = \boldsymbol{q}^{(n)}$
  2: Parse $k_b$ as $k_b^{(\mathsf{MIC})} || k_b^{(\mathsf{select})} || r_b^{(s)} || r_b^{(u)} || r_b^{(y)} || c_b$
  3: $\boldsymbol{t}_b \leftarrow \mathsf{Eval}^{\mathsf{MIC}}_{n,\boldsymbol{p},\boldsymbol{q}}(b, k_b^{(\mathsf{MIC})}, \hat{x})$
  4: $z_b = t_{b,0} \mod 2$
  5: $s_b = \sum_{i=n}^{2n-1} t_{b,i} \mod 2$
  6: $\hat{s}_b = s_b + r_b^{(s)}$
  7: $e_b = -126 \cdot t_{b,0} + (n - f - 1) \cdot t_{b,n} + \sum_{i=1}^{n-1} (t_{b,i} + t_{b,2n-i}) \cdot (i - f - 2)$
  8: $\hat{u}_b = r_b^{(u)} + t_{b,n} + \sum_{i=1}^{n-1} (t_{b,i} + t_{b,2n-i}) \cdot 2^{n-i}$
  9: $(\hat{s}, \hat{u}) =$ reconstruct $(\hat{s}_b, \hat{u}_b)$
  10: $\hat{y}_b \leftarrow 2 \cdot \mathsf{Eval}^{\mathsf{select}}_n(b, k_b^{(\mathsf{select})}, 1 - \hat{s}, \hat{x}) - b \cdot \hat{x}$
  11: $\hat{y} =$ reconstruct $(\hat{y}_b)$
  12: $m_b = \mathsf{TR}(b \cdot \hat{y} \cdot \hat{u} - r_b^{(y)} \cdot \hat{u} - r_b^{(u)} \cdot \hat{y} + c_b, n - 24)$
  13: **return** $(z_b, s_b, e_b, m_b)$
```

Fig. 7: Protocol for FixToFloat.

I. *FloatToFix*

For a given floating-point number $(z, s, e, m) \in \mathbb{FP}$, the functionality $\mathsf{FloatToFix}_{n,f}$ calculates a fixed-point number $x \in \mathbb{U}_N$ with precision f such that:

$$\llbracket x \rrbracket_{n,f} = (1 - z) \cdot (1 - 2s) \cdot 2^{\mathsf{int}_{10}(e)} \cdot \llbracket m \rrbracket^{+}_{24,23}$$
$$\implies x = (1 - z) \cdot (1 - 2s) \cdot \lfloor m \cdot 2^{\mathsf{int}_{10}(e) + f - 23} \rceil$$

But since we are only concerned with softmax output, we restrict the discussion to the case when $s = 0$, $z = 0$, and $\mathsf{int}_{10}(e) < 0$. Note that the protocol can be generalized to other cases trivially using calls to the protocol for select. So, we can simplify the above equation to:

$$x = \lfloor m \cdot 2^{\mathsf{int}_{10}(e')} \rceil$$

where $e' = e + f - 23$. Note that since $\mathsf{int}_{10}(e') < f - 23$, $\mathsf{int}_{10}(e')$ can take both positive or negative values[5]. Also, as m contains a 24-bit value, $x = 0$ when $\mathsf{int}_{10}(e') \leqslant -24$.

[5]In the special case when $f = 24$, $\mathsf{int}_{10}(e')$ is always non-positive.

So, for $m \in \mathbb{U}_{2^{24}}$ and $e \in \mathbb{U}_{2^{10}}$, let us define a functionality $\mathsf{adjust}_n : \mathbb{U}_{2^{24}} \times \mathbb{U}_{2^{10}} \to \mathbb{U}_N$ such that:

$$\mathsf{adjust}_n(m, e') = \begin{cases} 0 & \text{if } \mathsf{int}_{10}(e') \leqslant -24 \\ m \cdot 2^{\mathsf{int}_{10}(e')} & \text{if } \mathsf{int}_{10}(e') \geqslant 0 \\ m \gg_L (-\mathsf{int}_{10}(e')) & \text{otherwise} \end{cases}$$

Then, we can rewrite the expression for $\mathsf{FloatToFix}_{n,f}$ as:

$$x = \mathsf{adjust}_n(m, e')$$

Hence, to implement $\mathsf{FloatToFix}_{n,f}$, we need to devise a protocol for adjust_n. One way to do this would be to calculate the value of x for all possible values of e and then use a lookup-table protocol to get the correct value of x. However, this approach would incur a high online communication cost, as each of the output values of adjust_n needs to be reconstructed. We explain our alternative approach in two steps. First, we derive a unified formula for the offset function of adjust_n for a public value of e'. Then, we discuss techniques to support masked $\hat{e}'$ in the same formula.

Consider the case when $-24 < \mathsf{int}_{10}(e') < 0$ first. In this case, adjust_n is equivalent to a logical right-shift by $-\mathsf{int}_{10}(e')$. Using the expression for logical right-shift by a public scale $-e$ from [18], ignoring the one-bit term, we get:

$$\begin{aligned}
&\mathsf{adjust}_n^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(\hat{m}, e') = (\hat{m} \gg_L \mathsf{int}_{10}(-e')) \\
&\quad - (\mathsf{r}^{\mathsf{in}} \gg_L \mathsf{int}_{10}(-e')) + 2^{24 + \mathsf{int}_{10}(e')} \cdot 1\{\hat{m} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}} \\
&= \mathsf{adjust}_n(\hat{m}, e') - \mathsf{adjust}_n(\mathsf{r}^{\mathsf{in}}, e') \\
&\quad + \mathsf{adjust}_n(1, e' + 24) \cdot 1\{\hat{m} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}}
\end{aligned}$$

For the case when $0 \leqslant \mathsf{int}_{10}(e') < f - 23$, adjust_n is equivalent to mixed-bitwidth multiplication with $2^{\mathsf{int}_{10}(e')}$. Hence, the expression for adjust_n can be written as:

$$\begin{aligned}
&\mathsf{adjust}_n^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}(\hat{m}, e') = 2^{\mathsf{int}_{10}(e')} \cdot (\hat{m} - \mathsf{r}^{\mathsf{in}} + 2^{24} \cdot 1\{\hat{m} < \mathsf{r}^{\mathsf{in}}\}) \\
&\quad + \mathsf{r}^{\mathsf{out}} \\
&= \mathsf{adjust}_n(\hat{m}, e') - \mathsf{adjust}_n(\mathsf{r}^{\mathsf{in}}, e') \\
&\quad + \mathsf{adjust}_n(1, e' + 24) \cdot 1\{\hat{m} < \mathsf{r}^{\mathsf{in}}\} + \mathsf{r}^{\mathsf{out}}
\end{aligned}$$

Hence, we arrive at a formula that works for both cases. Note that this formula automatically handles the remaining case (i.e. $\mathsf{int}_{10}(e') \leqslant -24$) due to our definition of adjust_n. Now, we need to modify the above expression to work when e' is not public. We describe techniques to calculate the shares of the four important terms in the above expression, namely $\mathsf{adjust}_n(\hat{m}, e')$, $\mathsf{adjust}_n(\mathsf{r}^{\mathsf{in}}, e')$, $\mathsf{adjust}_n(1, e'+24)$ and $1\{\hat{m} < \mathsf{r}^{\mathsf{in}}\}$. The four terms can then be combined trivially using local additions and a call to the protocol for select.

Notice that for a given e', the terms $\mathsf{adjust}_n(\hat{m}, e')$ and $\mathsf{adjust}_n(1, e'+24)$ can be calculated locally. So, if the evaluators have secret shares of the one-hot vector of e', shares of the terms, $\mathsf{adjust}_n(\hat{m}, e')$ and $\mathsf{adjust}_n(1, e'+24)$ can be calculated locally by elementwise-multiplying the shares of the vector with the correct constant value for the corresponding e' value and adding them. To get the one-hot vector of e', we employ the following technique. Dealer sends shares of the one-hot vector of $2^{10} - r^{(e')}$, where $r^{(e')}$ is the random mask chosen for e'. Evaluators can then rotate this shared array to the right by $\hat{e'}$ to get the shares of the one-hot vector of e'. Similarly, to get shares of the term $\mathsf{adjust}_n(\mathsf{r}^{\mathsf{in}}, e')$, the dealer sends an array of shares of all possible values of $\mathsf{adjust}_n(\mathsf{r}^{\mathsf{in}}, e')$ rotated to the right by $r^{(e')}$. Evaluators can then simply index the secret-shared array at $\hat{e'}$ to get the shares of $\mathsf{adjust}_n(\mathsf{r}^{\mathsf{in}}, e')$.

```
FloatToFix $\Pi_{n,f}^{\mathsf{FloatToFix}}$
$\mathsf{Gen}_{n,f}^{\mathsf{FloatToFix}}(\mathsf{r}^{\mathsf{out}})$ :
  1: $r^{(m)} \xleftarrow{\$} \mathbb{U}_{2^{24}}$
  2: $r^{(e')} \xleftarrow{\$} \mathbb{U}_{1024}$
  3: $r^{(w)} \xleftarrow{\$} \mathbb{U}_{2}$
  4: $r^{(t)} \xleftarrow{\$} \mathbb{U}_{N}$
  5: $(k_0^<, k_1^<) \leftarrow \mathsf{Gen}_{24}^<(1^\lambda, r^{(m)}, 1, \mathbb{U}_2)$
  6: Let $\boldsymbol{p} = \{1\{i = 1024 - r^{(e')}\}\}_i \in \mathbb{U}_N^{1024}$.
  7: Let $\boldsymbol{q} = \{\mathsf{r}^{\mathsf{out}} - \mathsf{adjust}(r^{(m)}, i)\}_i \in \mathbb{U}_N^{1024}$.
  8: $\boldsymbol{q} = \boldsymbol{q} \ggg r^{(e')}$
  9: share $(r^{(m)}, r^{(e')}, r^{(w)}, r^{(t)}, \boldsymbol{p}, \boldsymbol{q})$
  10: $(k_0^{(\mathsf{select})}, k_1^{(\mathsf{select})}) \leftarrow \mathsf{Gen}_n^{\mathsf{select}}(r^{(w)}, r^{(t)}, 0)$
  11: For $b \in \{0,1\}$, $k_b = k_b^< || k_b^{\mathsf{select}} || r_b^{(m)} || r_b^{(e')} || r_b^{(w)} || r_b^{(t)} || \boldsymbol{p}_b || \boldsymbol{q}_b$

$\mathsf{Eval}_{n,f}^{\mathsf{FloatToFix}}(b, k_b, (m_b, e_b))$ :
  1: Parse $k_b$ as $k_b^< || k_b^{\mathsf{select}} || r_b^{(m)} || r_b^{(e')} || r_b^{(w)} || r_b^{(t)} || \boldsymbol{p}_b || \boldsymbol{q}_b$
  2: $\hat{m}_b = m_b + r_b^{(m)}$
  3: $\hat{e}'_b = e_b + r_b^{(e')} + b \cdot (f - 23)$
  4: $(\hat{m}, \hat{e}') = \mathsf{reconstruct}\ (\hat{m}_b, \hat{e}'_b)$
  5: $\hat{w}_b \leftarrow \mathsf{Eval}_{24}^<(b, k_b^<, \hat{m}) + r_b^{(w)} \mod 2$
  6: $\boldsymbol{p}'_b = \boldsymbol{p}_b \ggg \hat{e}'$
  7: $\hat{t}_b = r_b^{(t)} + \sum_{i=0}^{1023} \mathsf{adjust}_n(1, i + 24 \mod 1024) \cdot p'_{b,i}$
  8: $(\hat{w}, \hat{t}) = \mathsf{reconstruct}\ (\hat{w}_b, \hat{t}_b)$
  9: $\hat{x}_b = q_{b,\hat{e}'} + \mathsf{Eval}_n^{(\mathsf{select})}(b, k_b^{(\mathsf{select})}, \hat{w}, \hat{t}) + \sum_{i=0}^{1023} \mathsf{adjust}(\hat{m}, i) \cdot p'_{b,i}$
  10: return $\hat{x}_b$
```

Fig. 8: Protocol for FloatToFix.

Based on the above discussion, we present the protocol $\Pi_{n,f}^{\mathsf{FloatToFix}}$ for $\mathsf{FloatToFix}_{n,f}$ in Figure 8. Note that, unlike other protocols, $\Pi_{n,f}^{\mathsf{FloatToFix}}$ starts with secret shares, as SECFLOAT returns shares of m and e. We now provide the simulator-based security proof of the protocol.

Proof. For $b \in \{0,1\}$, the simulator Sim_b for FloatToFix is given (m_b, e_b) and $\hat{x}_b$ and has to simulate the view of party b, i.e., messages $k_b^< || k_b^{\mathsf{select}} || r_b^{(m)} || r_b^{(e')} || r_b^{(w)} || r_b^{(t)} || \boldsymbol{p}_b || \boldsymbol{q}_b$ from dealer and $\hat{m}_{1-b} || \hat{e}'_{1-b} || \hat{w}_{1-b} || \hat{t}_{1-b}$ from the other evaluator. The simulator follows the following steps:

1) Randomly samples $r_{b,\mathsf{sim}}^{(m)}$ and $\hat{m}_{1-b,\mathsf{sim}}$ from $\mathbb{U}_{2^{24}}$.
2) Randomly samples $r_{b,\mathsf{sim}}^{(e')}$ and $\hat{e}'_{1-b,\mathsf{sim}}$ from $\mathbb{U}_{2^{10}}$.
3) Sets $\hat{m}_{\mathsf{sim}} = m_b + r_{b,\mathsf{sim}}^{(m)} + \hat{m}_{1-b,\mathsf{sim}}$ and $\hat{e}'_{\mathsf{sim}} = e_b + r_{b,\mathsf{sim}}^{(e')} + \hat{e}'_{1-b} + b \cdot (f - 23)$.
4) Randomly samples $r_{b,\mathsf{sim}}^{(w)}$, $\hat{w}_{b,\mathsf{sim}}$ and $\hat{w}_{1-b,\mathsf{sim}}$ from $\mathbb{U}_2$.
5) Generates $k_{b,\mathsf{sim}}^<$ using $\mathsf{Sim}_b^<$ with input $\hat{m}_{\mathsf{sim}}$ and output $\hat{w}_{b,\mathsf{sim}} - r_{b,\mathsf{sim}}^{(w)}$.
6) Randomly samples $\boldsymbol{p}_{b,\mathsf{sim}}$ from $\mathbb{U}_N^{1024}$.
7) Sets $\boldsymbol{p}'_{b,\mathsf{sim}} = \boldsymbol{p}_{b,\mathsf{sim}} \ggg \hat{e}'_{\mathsf{sim}}$.
8) Randomly samples $r_{b,\mathsf{sim}}^{(t)}$ and $\hat{t}_{1-b,\mathsf{sim}}$ from $\mathbb{U}_N$.
9) Calculates $\hat{t}_{b,\mathsf{sim}}$ using step 7 from $\mathsf{Eval}_{n,f}^{\mathsf{FloatToFix}}$.
10) Sets $\hat{w}_{\mathsf{sim}} = \hat{w}_{b,\mathsf{sim}} + \hat{w}_{1-b,\mathsf{sim}}$
11) Sets $\hat{t}_{\mathsf{sim}} = \hat{t}_{b,\mathsf{sim}} + \hat{t}_{1-b,\mathsf{sim}}$.
12) Generates $k_{b,\mathsf{sim}}^{\mathsf{select}}$ using $\mathsf{Sim}_b^{\mathsf{select}}$ with input $(\hat{w}_{\mathsf{sim}}, \hat{t}_{\mathsf{sim}})$ and output h randomly sampled from $\mathbb{U}_N$.
13) Calculates $q_{b,\mathsf{sim},\hat{e}'_{\mathsf{sim}}} = \hat{x}_b - h - \sum_{i=0}^{1023} \mathsf{adjust}(\hat{m}_{\mathsf{sim}}, i) \cdot p'_{b,\mathsf{sim},i}$.
14) For $k \in [0, 1023] - \{\hat{e}'_{\mathsf{sim}}\}$, randomly samples $q_{b,\mathsf{sim},k}$ and sets $\boldsymbol{q}_{b,\mathsf{sim}} = \{q_{b,\mathsf{sim},i}\}_i$.

□

Note. The key size of $\Pi_{n,f}^{\mathsf{FloatToFix}}$ can be further reduced by only sending the elements in arrays $\boldsymbol{p}$ and $\boldsymbol{q}$ which needs to be accessed in the evaluation, based on the constraints on the value of e. Moreover, as the array $\boldsymbol{p}$ is set to 1 at a single index, it can be replaced with a Distributed Point Function key [19] to further reduce the key size at the cost of increased compute. We omit these optimizations for simplicity.

J. End-to-end training protocols

Given the protocols discussed in the previous sections, we now discuss how they can be stitched together to obtain an end-to-end protocol for securely computing any function. Let us assume that the cleartext function comprises of two functionalities $A : \mathbb{G}_1 \to \mathbb{G}_2$ and $B : \mathbb{G}_2 \to \mathbb{G}_3$ that are sequentially invoked and let us assume that their corresponding secure protocols are Π^A and Π^B, respectively. To devise a secure protocol for $B(A(x))$, the dealer simply sets the input random mask of Π^B equal to the output random mask of Π^A and runs the respective Gen algorithms to generate the FSS keys. The evaluators can then pass the output of $\hat{\Pi}^A$ to the input of $\hat{\Pi}^B$ to get the required masked output. Arbitrary number of protocols can also be composed in a similar way. The security of this protocol can be argued in the simulation paradigm as follows. Since each individual protocol returns values masked by a random value, it suffices to set these intermediate values to a random value (from the corresponding group) during simulation. Simulators for the constituent protocols can then be sequentially invoked to complete the simulation of the overall protocol.

Now we consider the case of fixed-point training. Let the global fixed-point scale be f. A typical machine learning model contains layers like convolution, matrix multiplication, ReLU, MaxPool, and so on, where convolutions and matrix multiplications are followed by truncations. A single training iteration constitutes three steps: forward pass, softmax calculation and backward pass. Once we have a protocol for a single iteration that outputs updated masked weights, multiple iterations can be trivially composed.

Forward Pass. Any sequence of the form convolution-ReLU-MaxPool is computed securely using the protocol for convolution followed by $\Pi^{\mathsf{TRM}}_{n,f}$. Sequences of the form convolution-ReLU in the remaining layers are computed securely using the protocol for convolution followed by $\Pi^{\mathsf{stTrReLU}}_{n,f}$. Similar computations are done for matrix multiplications (in place of convolutions) as well. For the remaining truncation and ReLU layers, we use the protocols $\Pi^{\mathsf{stTr}}_{n,f}$ and Π^{ReLU}_{n}, respectively.
Softmax. The masked fixed-point outputs of the forward pass protocol are then converted into secret-shared floating point numbers using the protocol $\Pi^{\mathsf{FixToFloat}}_{n,f}$. These secret shares are then passed into the protocol for softmax from SECFLOAT, which returns the output as a secret shared floating point numbers, which are then converted back to masked fixed-point numbers using $\Pi^{\mathsf{FloatToFix}}_{n,f}$. The masked one-hot vector of the training label is then locally subtracted from the output to calculate the required input to the backward pass.
Backward Pass. The outputs of DReLU that were computed during the forward pass can be reused during the backward pass; ReLU is hence realized using a call to $\Pi^{\mathsf{select}}_{n}$, while for MaxPool, the protocol for bitwise-AND and $\Pi^{\mathsf{select}}_{n}$ suffices (along with the stored forward pass values). Hence there is no benefit in fusing layers as was done in the forward pass. Backward pass of convolution and matrix multiplication are realized using the protocols for convolutions, matrix multiplications and $\Pi^{\mathsf{stTr}}_{n,f}$.

K. Piranha Functionalities using FSS protocols

In this section, we show how to build FSS-based protocols for functionalities used by PIRANHA [64]. These functionalities are implemented in our library in order to directly compare the performance of ORCA with PIRANHA on the same benchmarks.

1) Local-Truncation + ReLU: As discussed several prior works on secure training use local truncations, i.e., parties locally truncate their secret shares. In the FSS setting, this translates to the dealer truncating the mask and the evaluators truncating the masked value. While comparing with these works, to be apples-to-apples, we provide FSS-based protocols and optimizations for this setting. In particular, our ideas of fusing truncations will non-activations such as ReLU or ReLU+Maxpool are also applicable for local truncations and result in lower key size compared to the naive approach of sequential computation.

A straightforward way to realize a protocol for Local-Truncation + ReLU for an n-bit input x with public scale f, with an error similar to the local truncation protocol [47], is to locally truncate x by f and use the result as an input to the protocol for ReLU from [18]. Over this protocol, we perform two optimizations:

1) We replace the single-round protocol for ReLU from [18] with the two-round protocol (Section V-A) with a smaller key size.
2) Since the value $x \gg_A f$ can be represented accurately in $(n-f)$-bits, we optimize the comparisons required for the calculation of DReLU bit to be done over a reduced bitwidth of $(n-f)$.

Using the above-described optimizations, we present the protocol for Local-Truncate + ReLU in Figure 9 and summarize its costs in the following theorem:

Local-Truncate + ReLU $\Pi^{\mathsf{localTrReLU}}_{n,f}$ $(\mathbb{G}^{\mathsf{out}} = \mathbb{U}_N)$

```
$\mathsf{Gen}^{\mathsf{localTrReLU}}_{n,f}(\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$ :
  1: $r' = \mathsf{r}^{\mathsf{out}} \gg_L f$
  2: $(k^<_0, k^<_1) \leftarrow \mathsf{Gen}^<_{n-f}(1^\lambda, r', 1, \mathbb{U}_2)$
  3: $r^{(d)} \xleftarrow{\$} \mathbb{U}_2$
  4: share $r^{(d)}$
  5: $(k^{(\mathsf{select})}_0, k^{(\mathsf{select})}_1) \leftarrow \mathsf{Gen}^{\mathsf{select}}_n(r^{(d)}, r', \mathsf{r}^{\mathsf{out}})$
  6: For $b \in \{0,1\}, k_b = k^<_b || r^{(d)}_b || k^{(\mathsf{select})}_b$

$\mathsf{Eval}^{\mathsf{localTrReLU}}_{n,f}(b, k_b, \hat{x})$ :
  1: Parse $k_b$ as $k^<_b || r^{(d)}_b || k^{(\mathsf{select})}_b$
  2: $\hat{x}' = \hat{x} \gg_L f$
  3: $\hat{y}' = \hat{x}' + 2^{n-f-1} \mod 2^{n-f}$
  4: $t_b \leftarrow \mathsf{Eval}^<_{n-f}(b, k^<_b, \hat{y}') - \mathsf{Eval}^<_{n-f}(b, k^<_b, \hat{x}')$
  5: $\hat{d}_b = t_b + b \cdot 1\{\hat{y}' \geqslant 2^{n-f-1}\} + r^{(d)}_b \mod 2$
  6: $\hat{d} = $ reconstruct $(\hat{d}_b)$
  7: return $\hat{u}_b \leftarrow \mathsf{Eval}^{\mathsf{select}}_{n-f}(b, k^{(\mathsf{select})}_b, \hat{d}, \hat{x}')$
```

Fig. 9: Protocol for Local-Truncate + ReLU. b refers to the party id.

Theorem 11. *$\Pi^{\mathsf{localTrReLU}}_{n,f}$ realizes local-truncation + ReLU securely such that $\mathsf{keysize}(\Pi^{\mathsf{localTrReLU}}_{n,f}) = \mathsf{keysize}(\mathsf{DCF}_{n-f,1}) + \mathsf{keysize}(\Pi^{\mathsf{select}}_n) + 1$. In the online phase, the protocol requires 2 evaluations of $\mathsf{DCF}_{n-f,1}$ and costs communication of 2 bits in 1 round.*

The naive unoptimized approach costs a key size of $\mathsf{keysize}(\mathsf{DCF}_{n,2n}) + 5n$ bits. For $n = 64$ and $f = 24$, our protocol costs $3\times$ less key size. In the case when local-truncation is followed by MaxPool followed by ReLU, we follow the idea described in Appendix F to calculate all the internal DReLU bits on a reduced bitwidth of $n - f$.

2) Approximate Softmax: PIRANHA used Equation 1 to implement softmax with the following approximations for exponential and inverse.

$$\tilde{\mathsf{exp}}(x) = \begin{cases} \frac{x+2}{2} & \text{if } -2 < x \leq 0 \\ 0 & \text{if } x \leq -2 \end{cases}$$
$$= \mathsf{relu}(x+2)/2$$

$$\tilde{\mathsf{inv}}(x) = \begin{cases} 2.63x^2 - 5.857x + 4.245 & \text{if } 0.5 \leq x < 1 \\ \tilde{\mathsf{inv}}(x/2^k) \cdot 2^{-k} & \text{if } 2^{k-1} \leqslant x < 2^k \end{cases}$$

The expression for $\tilde{\mathsf{exp}}(x)$ can be trivially realized using the protocol for local-truncation + ReLU (Section K1). For $\tilde{\mathsf{inv}}(x)$, PIRANHA calculated k securely but revealed its value in the clear, leaking the range of x. To mimic this behaviour, we first calculate k using the protocol for multiple interval containment (Section IV-C) and reveal its value to the evaluators. Then the above polynomial can be trivially evaluated using protocols for multiplication and local-truncation.