Orca: FSS-based Secure Training and Inference with GPUs

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

Kanav Gupta* Microsoft Research, India kanav0610@gmail.com

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

Nishanth Chandran Microsoft Research, India nichandr@microsoft.com

Divya Gupta Microsoft Research, India divya.gupta@microsoft.com

Rahul Sharma Microsoft Research, India rahsha@microsoft.com

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00063

***Abstract*—Secure Two-party Computation (2PC) allows two parties to compute any function on their private inputs without revealing their inputs to each other. In the offline/online model for 2PC, correlated randomness that is independent of all inputs to the computation, is generated in a preprocessing (offline) phase and this randomness is then utilized in the online phase once the inputs to the parties become available. Most 2PC works focus on optimizing the online time as this overhead lies on the critical path. A recent paradigm for obtaining efficient 2PC protocols with low online cost is 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 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, $98\times$ lesser communication, and is $22\times$ faster on CIFAR-10. For secure ImageNet inference, ORCA achieves sub-second latency for VGG-16 and ResNet-50 and outperforms the state-of-the-art by $8-103\times$.

1. 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) [29], [67] provide secure training over this sensitive data [40], [47], [48], [59], [61], [63], [65]. 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 [48], to months [63], to weeks [59], to a day with PIRANHA [65]. PIRANHA is the current state-of-the-art in accelerating secure training using GPUs. Our goal is to reduce this time to less than an hour.

PIRANHA works in the *offline/online model* where the data-independent correlated randomness is generated in an offline pre-processing phase, and parties use this correlated randomness 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 [59], using local truncations [48], [59] and approximating exponentiations with piecewise-linear functions. Moreover, following Falcon [63], it also trade-offs security for efficiency, and reveals intermediate values during softmax computation, 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. For example, PIRANHA reports that VGG16 on CIFAR-10 gives 67% accuracy in PyTorch, which drops to 55% in secure training [65].

1.1. Our Contributions

Unlike [48], [59], [61], [63], [65], our secure training remains faithful to quantized training algorithms from ML literature [31] that are known to mirror floating-point PyTorch training. We show that faithful secure training of small models (with thousands of parameters) produces more accurate models than secure training of large models (with millions of parameters) with PIRANHA. PIRANHA approximations fail on small models - training our small models with these approximations (either local truncations or piecewise-linear approximation of exponentiation) reduces the accuracy of the trained model to that of a random classifier. On CIFAR-10, our secure training beats PIRANHA on all metrics. We

. * Equal contribution.
. † Work partially done while at Microsoft Research.

are more secure (as we don't reveal any intermediate value) and produce models with 4% higher accuracy in $22\times$ less time while incurring $98\times$ lower communication. ORCA reaches a CIFAR-10 accuracy of 59.6% in 52 minutes and 69% in 128 minutes (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 pre-processing model [16], [19], [30]. 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 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.

1.1.1. System optimizations (detailed in Section 3). To accelerate compute, we create efficient GPU implementations of FSS protocols 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 [57]. The key to our accelerated implementation is a unique combination of system optimizations guided by the GPU micro-architecture. 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. 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 FSS key size for commonly occurring nodes in training.

1.1.2. Cryptographic improvements. Staying faithful to quantized training, e.g., the one described in Gupta et al. [31], requires efficient protocols for *stochastic truncations* of fixed-point values, ReLUs, maxpools, and floating-point softmax. Stochastic truncations, ReLUs and maxpools are expensive and lead to large keys. We create novel protocols that reduce the key size by up to $5\times$ (Section 5) that are crucial for obtaining low end-to-end latencies. Quantized training performs almost all operations in fixed-point but uses floating-point arithmetic in softmax computations to maintain accuracy [31]. In secure training literature, there are three ways of computing exponentiations occurring in softmax. First, use the very cheap piecewise-linear approximations [41], [65]. Second, use fixed-point approximations that are more expensive [39], [40]. Third, use even more expensive but precise computation over floating-point [53]. To stay faithful to the computations done by Gupta et al. [31], 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 6).

1.1.3. ORCA. We implement our techniques in ORCA, a push-button tool for secure training, that will be made publicly available[1]. ORCA sports a library of GPU-accelerated FSS blocks that future research in this area can build upon (Section 7). We show that training small models with ORCA while staying faithful (see Figure 5) to quantized training [31], outperforms the approximate large model training of PIRANHA in accuracy (upto 14%), time $(4-22\times)$, and communication $(39-98\times)$ (Section 8.1). In a more apples-to-apples comparison, ORCA outperforms PIRANHA on approximate training of large models by $\approx 7.3\times$ (Section 8.2.1). ORCA also outperforms prior works that don't use PIRANHA approximations by $7-693\times$ (Section 8.2.2). ORCA's protocols need smaller FSS keys (Table 8) and ORCA's GPU-based protocols outperform their CPU counterparts by an order of magnitude (Table 7). Finally, ORCA achieves the first ever sub-second ImageNet-scale inference of VGG-16 and ResNet-50 and outperforms the state-of-the-art [30], [41] by $8-103\times$ (Table 9).

2. Preliminaries

2.1. 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 $\boldsymbol{x} = \{x_0, x_1, \dots\}$.

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 $x = \lfloor r \cdot 2^f \rfloor \bmod N$. 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. $\frac{\mathsf{int}_n(x)}{2^f}$).

Operators. Arithmetic operations in $\mathbb{U}_N$, like addition and multiplication, are followed by a $\bmod N$ 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 $\mathsf{TR}(x, f)$,

1. https://github.com/mpc-msri/EzPC.git

is defined as dropping the lower f bits of input and returning the output as an $(n-f)$-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.

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 $b \in \{0,1\}$.

2.2. Threat Model

We consider 2-party secure computation (2PC) in the pre-processing model [30], [41], [57], [65]. In the pre-processing phase, correlated randomness independent of all inputs to the computation is generated. This can be generated in several ways – through a trusted dealer [16], [19], [30], [41], [57], [65], a 2PC protocol [29], [67], or through more efficient specialized 2PC protocols [26]. In this work, we deploy the first method. We prove the security of our protocols in the standard simulation paradigm [20], [29], [45] against a semi-honest static probabilistic polynomial time (PPT) adversary that corrupts one of the two parties.

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 [48]. 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.

2.3. Function Secret Sharing and DCFs

A Function Secret Sharing (FSS) Scheme [17], [18] 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 requires that each function g_b hides g.

Definition 1 (FSS: Syntax [17], [18])**.** *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. Size of k_0 or k_1 is the *key size* and corresponds to the correlated randomness required to be stored by each evaluator.

Distributed Comparison Functions (DCFs) were introduced by Boyle et al. [18] and provide an FSS scheme for special interval functions.

Definition 2 (DCF [16], [18])**.** *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 [16].

Theorem 1 (Cost of DCF [16])**.** *Given 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* $n(\lambda+\ell+2)+\lambda+\ell$*. Number of PRG invocations in* $\mathsf{Gen}^<_n$ *is* $2n$ *and in* $\mathsf{Eval}^<_n$ *is* n.

We write $\mathsf{DCF}_{n,\ell}$ and $\mathsf{keysize}(\mathsf{DCF}_{n,\ell})$ to represent this scheme and its key size, respectively. Similar to prior works [16], [30], we set $\lambda = 126$ and realize the required PRG using 4 calls to AES with 128-bit output in CTR mode. Next, as observed in [30], during evaluation it suffices to make only 2 AES calls because only 2 out of 4 blocks are needed in the evaluation algorithm of [16] and CTR mode allows us to only generate the required blocks. When we require a DCF with smaller output length, we use the *early termination optimization* from [18] (ported to the context of DCFs) and obtain the following costs.

Theorem 2 (DCF with small payloads)**.** *For* $i \in \{0,1,2,3,4,5,6\}$*,* $\ell = 2^i$*,* $\lambda = 126$ *and* $n > 7-i$*, the key size of* $\mathsf{DCF}_{n,\ell}$ *is* $(n-8+i)(\lambda+\ell+2)+\lambda+256$*. The number of AES calls in* $\mathsf{Gen}^<_n$ *is* $8(n-8+i)+4$ *and in* $\mathsf{Eval}^<_n$ *is* $2(n-8+i)+1$*. When* $n \leq 7-i$*, the key size of* $\mathsf{DCF}_{n,\ell}$ *is* 2^{n+i} *and the number of AES calls in* $\mathsf{Gen}^<_n$ *and* $\mathsf{Eval}^<_n$ *is* 0*.*

2.4. Secure 2PC with pre-processing using FSS

[16], [19] proposed a semi -honest static secure 2PC protocol using FSS. Consider two evaluators who want to evaluate the computation circuit with gates $\{g_i\}_i$ and wires $\{w_i\}_i$. The protocol evaluates this circuit securely in two phases - offline and online.

2.4.1. Offline Phase. For each wire w_i in the computation circuit, randomly sample a mask r_i. For each gate g, with input wire w_i and output wire w_j, generate FSS keys (k^g_0, k^g_1) for the *offset function*, $g^{[r_i,r_j]}(x) = g(x-r_i)+r_j$, and provide party b with k^g_b. For input and output wires w_i owned by party b, party b learns the corresponding mask r_i.

2.4.2. 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 from the preprocessing phase to obtain output in clear.

2.5. 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, and graph processing. Even a mid-range NVIDIA A6000 GPU can execute over five thousand threads concurrently. 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; threads from different threadblocks can run on different SMs.

GPU's memory subsystem reflects a similar hierarchy. The memory onboard the GPU typically supports more than TB/sec bandwidth but is 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 allocate/de-allocate 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.

3. 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.

3.1. Accelerating FSS-based compute on GPU

The prior work that attempted to accelerate FSS-based protocol on GPU [57] observed only modest speedups 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.

① Faster AES computation (AES) A key primitive in FSS-based computation is the Distributed Comparison Function or DCF (Section 4). We empirically find that computing DCF can account for the majority of overall computation time, e.g., in the forward pass of CNN3, DCFs account for about 93% of the overall compute time. Evaluating a DCF over $x \in \mathbb{U}_N$ requires $2n$ invocations of AES (Section 2.3).

Consider the task of performing 10 million DCF evaluations for $n = 64$ 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 1 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 [57], [59]. AES requires repeatedly looking up precomputed lookup tables [60]. 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 indices (thus, different addresses).

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 [60]. 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 1 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.
② Optimized data layout for cache locality (LAYOUT) Computing a DCF requires the evaluator to perform n *chained* PRG ($2n$ AES) [16], [30]. 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) [16]. 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 contiguously in the memory is better for 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 improves the L1 cache hit rate from 20% to 49%. As shown in Table 1, 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 (Section 3.2). We address this challenge with novel cryptographic protocols that significantly reduce the keysize for commonly occurring nodes in training and inference detailed in Sections 4, 5 and 6. A reduction in keysize not only reduces the data movement between the CPU and GPU over the slow PCIe interconnect, it also reduces the footprint on limited GPU memory. A key technique in reducing keysize 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.

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 [64]. 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 1 shows further speedup due 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.

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

Table 1: Speedup of 10M DCFs with our optimizations

3.2. 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, CNN3 from [31] needs 34.7 GB of key per iteration prior to cryptographic improvements. Bypassing the OS's page cache, reduces the time to read this key from 13.4s to 6.2s.
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 SSD can overshadow the GPU computation time for 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 the GPU-based implementation without the optimizations in Section 3.1.

We construct new FSS protocols to reduce keysize in popular non-linearities like ReLU and maxpool while reducing online communication as well when combined with truncations in quantized training (Sections 4,5). For instance, we get $5.7\times$ reduction in keysize for truncation followed by ReLU. Overall, for training, we get up to $3\times$ reduction in key size (Table 8). For CNN3, we reduce the time to read the key per iteration from 6.2s to 1.9s that can now be perfectly overlapped with 2.4s of GPU compute (Table 12).

4. Protocols for Basic Building Blocks

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 mask r^{in} (generated in the preprocessing phase) is unknown to the evaluators. 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 reconstruct. We denote the key size required by the protocol Π (or, $\hat{\Pi}$) by $\mathsf{keysize}(\Pi)$.

Unlike prior works on FSS-based 2PC [16], [19], [30] where the online phase for a gate was non-interactive, we construct more complex protocols, where the online phase is allowed to be multi-round. However, this poses no issue w.r.t. stitching together of protocols as evaluators still end with masked values or shares of the same.

Below, we first provide new FSS-based protocols for functions Select and Signed Extension that would be used as building blocks in the protocols described in later sections.

4.1. Select

Select function, $\mathsf{select}_n : \{0,1\} \times \mathbb{U}_N \to \mathbb{U}_N$, takes as input a selector bit $s \in \{0,1\}$ and an n-bit payload $x \in \mathbb{U}_N$, 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 [30], 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 3. *Π_n^{select} in Figure 1 realises select_n securely with $\mathsf{keysize}(\Pi_n^{\mathsf{select}}) = 4n$ and no communication.*

```
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: $w = u\cdot\mathsf{r}_2^{\mathsf{in}} + \mathsf{r}^{\mathsf{out}}$
  3: $z = 2\cdot u\cdot\mathsf{r}_2^{\mathsf{in}}$
  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
```

Figure 1: Protocol for Select.

4.2. Signed Extension

[30] provides the following expression for the offset function of SignExt functionality (Section 2):

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

where $\hat{x}' = \hat{x} + 2^{m-1} \mod 2^m$. [30] 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 bit, we achieve a smaller key size compared to [30] due to a smaller 1-bit payload (and early termination).

```
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: $\hat{x}' = \hat{x} + 2^{m-1} \mod 2^m$
  3: $\hat{w}_b \leftarrow \mathsf{Eval}_m^<(b, k_b^<, \hat{x}') + r_b^{(w)} \mod 2$
  4: $\hat{w} = \mathsf{reconstruct}\ (\hat{w}_b)$
  5: return $\hat{y}_b = b\cdot\hat{x}' + p_{b,\hat{w}}$
```

Figure 2: Protocol for SignExt.

Theorem 4. *$\Pi_{m,n}^{\mathsf{SignExt}}$ 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.*

5. 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 [30] (i.e., through the use of Beaver triples generated in the preprocessing phase). 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 [31], [32], [40], and local truncations [48]. Most prior works on secure training [48], [59], [61], [65] use local truncations as they are most efficient due to being local operations. LLAMA [30] provides an FSS protocol for faithful truncation. Gupta et al. [31] make a case for the necessity of stochastic truncations in quantized training, and we provide FSS protocols for stochastic truncation (defined in Section 5.2). 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.

Section Overview. We begin by describing our new protocols for ReLU (Section 5.1) and stochastic truncation (Section 5.2) with reduced key size. Next, in Section 5.3, we show how to fuse the stochastic truncation nodes with activation functions such as ReLU and ReLU+Maxpool to obtain lower key size and compute compared to the naïve approach of sequential computation. Throughout, our focus is on reducing the key size (even if we slightly increase the number of online rounds of communication).

5.1. ReLU

For a signed value x, the ReLU functionality returns $\max(x,0)$. AriaNN [57] provides a 1-round protocol for ReLU_n (with a 1-bit error) with a key size of $\approx (n+1)(\lambda+2n)$, while [16] 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 $(n-8)(\lambda+3)+\lambda+4n+257$ and 2 additional bits of online communication compared to [16]. Concretely, for $n=64$, this results in a $\approx 2.1\times$ reduction in key size.

When $x \in \mathbb{U}_N$ is an n-bit 2's-complement representation of an underlying signed value,

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

where $\mathsf{DReLU}_n(x) = 1\{x < 2^{n-1}\}$ is called Derivative of ReLU. Using this expression, we calculate $\mathsf{DReLU}_n(x)$ in the first round with a single bit output and then use 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 [16] with comparison output in $2n$ bits.

```
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 $\hat{y}_b = v_b - u_b + b \cdot 1\{\hat{y} \geqslant 2^{n-1}\} + \mathsf{r}_b^{\mathsf{out}} \mod 2$
```

Figure 3: Protocol for DReLU.

We prove the following expression for the offset function of DReLU_n (in full version [34]): 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}.

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.*

5.2. Stochastic Truncations

In stochastic truncations, both bitwidth preserving and bitwidth reducing, the result of truncation is rounded up or down, with a probability depending on the truncated fractional part. We provide novel secure protocols for both of these versions below.

5.2.1. Stochastic Truncate-Reduce.

Definition 3. *For $x \in \mathbb{U}_N$, $x_0 = x \mod 2^f$, the stochastic truncate-reduce of x with f, represented by $\mathsf{stTR}(x,f) \in \mathbb{U}_{2^{n-f}}$ is y such that*

$$y = \begin{cases} \mathsf{TR}(x,f) & \textit{with probability} \quad 1 - x_0 \cdot 2^{-f} \\ \mathsf{TR}(x,f)+1 & \textit{with probability} \quad x_0 \cdot 2^{-f} \end{cases}$$

Now, $\mathsf{stTR}(x,f)$ can be computed as follows: Pick a value w at random that is 1 with probability $x_0 \cdot 2^{-f}$ and 0 otherwise, and output $\mathsf{TR}(x,f)+w$. Next, we observe that w can be computed as $1\{s < x_0\}$ for a random $s \in \mathbb{U}_{2^f}$.

We prove the following lemma (in the full version [34]) that allows us to provide a secure protocol for stochastic truncate-reduce at the cost of $\mathsf{DCF}_{f,1}$.

Lemma 1. *For a masked value $\hat{x} \in \mathbb{U}_N$ with underlying value x and random mask $r^{(x)}$, let $x_0 = x \mod 2^f$, $\hat{x_0} = \hat{x} \mod 2^f$ and $r_0^{(x)} = r^{(x)} \mod 2^f$. Then,*

$$\mathsf{TR}(x, f) = \mathsf{TR}(\hat{x}, f) - \mathsf{TR}(r^{(x)}, f) - 1\{\hat{x}_0 < r_0^{(x)}\}$$
$$1\{s < x_0\} = 1\{\hat{x}_0 < r_0^{(x)}\} - 1\{\hat{s} < r_0^{(x)}\} + 1\{\hat{s} < \hat{x_0}\}$$

And, hence,

$$\begin{aligned}\mathsf{stTR}(x, f) &= \mathsf{TR}(x, f) + 1\{s < x_0\}\\ &= \mathsf{TR}(\hat{x}, f) - \mathsf{TR}(r^{(x)}, f) - 1\{\hat{s} < r_0^{(x)}\} + 1\{\hat{s} < \hat{x_0}\}\end{aligned}$$

For the final expression in the above lemma, the first term can be computed locally by the evaluators, the second and the third terms can be computed by the dealer, and the last term can be computed using a $\mathsf{DCF}_{f,1}$. Interestingly, we also note, from the above lemma, that (faithful) truncate-reduce and stochastic truncate-reduce can both be computed with a similar cost of a single $\mathsf{DCF}_{f,1}$. We provide a formal protocol for stochastic truncate-reduce, $\Pi^{\mathsf{stTR}}_{n,f}$, and its security proof in Appendix C, with cost summarized below.

Theorem 6. *$\Pi^{\mathsf{stTR}}_{n,f}$ realizes stochastic truncate-reduce securely such that $\mathsf{keysize}(\Pi^{\mathsf{stTR}}_{n,f}) = \mathsf{keysize}(\mathsf{DCF}_{f,1}) + 2(n - f)$ bits. The online phase requires 1 evaluation of $\mathsf{DCF}_{f,1}$ and communication of 2 bits in 1 round.*

5.2.2. Stochastic Truncation.

Definition 4. *For $x \in \mathbb{U}_N$, $x_0 = x \mod 2^f$, the stochastic-truncation of x by f, represented by $x \gg_{\mathsf{st}} f$, is an n-bit number $y \in \mathbb{U}_N$ such that:*

$$y = \begin{cases}(x \gg_A f) & \text{with probability} \quad 1 - x_0 \cdot 2^{-f}\\ (x \gg_A f) + 1 & \text{with probability} \quad x_0 \cdot 2^{-f}\end{cases}$$

Using similar ideas as above, one way to compute stochastic truncation is $x \gg_{\mathsf{st}} f = (x \gg_A f) + 1\{s < x_0\}$, for a random s in $\mathbb{U}_{2^f}$. Here, the first term, $(x \gg_A f)$ can be computed using the arithmetic right shift protocol from Boyle et al. [16] and the second term, $1\{s < x_0\}$ can be computed using the expression in Lemma 1. Overall this approach would be require a $\mathsf{DCF}_{n-1,2n}$, a $\mathsf{DCF}_{f,n}$ and 2 $\mathsf{DCF}_{f,1}$, which is expensive. We prove the following lemma (in the full version [34]) that allows us to securely realize stochastic truncation at the cost of a stochastic truncate-reduce by f and a signed extension from $n - f$ to n bits[2].

Lemma 2. *For $x \in \mathbb{U}_N$ such that $\mathsf{int}_n(x) \leqslant 2^{n-1} - 2^f$,*

$$x \gg_{\mathsf{st}} f = \mathsf{SignExt}_{n-f,n}(\mathsf{stTR}(x, f))$$

Protocol $\Pi^{\mathsf{stTr}}_{n,f}$. Parties first run a protocol for stochastic truncate-reduce by f followed by a reconstruction of masked output, i.e., $\hat{\Pi}^{\mathsf{stTR}}_{n,f}$, followed by a protocol for signed extension to n-bits, i.e., $\Pi^{\mathsf{SignExt}}_{n-f,n}$.

2. We note that it is common to assume that the largest values occurring in the computation can be represented in lesser number of bits than the bitwidth used. Nonetheless, to ensure that security holds for arbitrary values occurring at runtime, we change our cleartext/ideal functionality to also use the expression on the right in Lemma 2.

Theorem 7. *$\Pi^{\mathsf{stTr}}_{n,f}$ realizes stochastic-truncation securely such that $\mathsf{keysize}(\Pi^{\mathsf{stTr}}_{n,f}) = \mathsf{keysize}(\Pi^{\mathsf{stTR}}_{n,f}) + \mathsf{keysize}(\Pi^{\mathsf{SignExt}}_{n-f,n})$. In the online phase, the protocol requires 1 evaluation each of $\hat{\Pi}^{\mathsf{stTr}}_{n,f}$ and $\Pi^{\mathsf{SignExt}}_{n-f,n}$ and communicates $2(n - f + 1)$ bits in 3 rounds.*

Compared to the straightforward approach described above, for $n = 64, f = 24$, our key size is $3.6\times$ lower and number of AES calls in online phase is $2.4\times$ lower.

5.3. 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 B, 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 5-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 2, in stochastic truncation, stochastic truncate-reduce by f bits is followed by a signed extension to n-bits. Next, 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 both the key size and online compute. Moreover, since ReLU output is always non-negative, we can use zero extension, whose protocol is very similar to signed extension (Section 4.2).

Second, we improve upon the above by providing a new protocol that does ReLU and zero extension together by leveraging similar comparisons done for DReLU and ZeroExt. This reduces the key size further. 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 $x \in \mathbb{U}_{2^{n-f}}$, mask r^{in} and masked value $\hat{x}$, let $d = \mathsf{DReLU}(x)$ and $w = 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\}$. Moreover, from [30],

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

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 $r^{(d)} \mod 2 = 1$. 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 F. Using this, we can trivially obtain a protocol for $\mathsf{stTrReLU}_{n,f}$ by running $\hat{\Pi}^{\mathsf{stTR}}_{n,f}$ followed by $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$.

Theorem 8. *$\Pi^{\mathsf{stTrReLU}}_{n,f}$ realizes $\mathsf{stTrReLU}_{n,f}$ securely such that $\mathsf{keysize}(\Pi^{\mathsf{stTrReLU}}_{n,f}) = \mathsf{keysize}(\Pi^{\mathsf{stTR}}_{n,f}) + \mathsf{keysize}(\mathsf{DCF}_{n-f,2}) + 6n + 4$. Online phase requires 1 evaluation of $\Pi^{\mathsf{stTR}}_{n,f}$ and 2 evaluations of $\mathsf{DCF}_{n-f,2}$ and communicates of $2(n - f + 5)$ bits in 3 rounds.*

Cost Comparison. We argue that our new approach to build fused protocols results in lower costs. The naive way of implementing this functionality (via protocols from LLAMA [30]) is by using truncation followed by ReLU and requires a key size of $\mathsf{keysize}(\mathsf{DCF}_{n-1,2n}) + \mathsf{keysize}(\mathsf{DCF}_{f,n}) + 2 \cdot \mathsf{keysize}(\mathsf{DCF}_{f,1}) + \mathsf{keysize}(\mathsf{DCF}_{n,2n}) + 9n$. On the other hand, the straightforward (non-fused) approach that uses our improved protocols has a key size of $\mathsf{keysize}(\mathsf{DCF}_{f,1}) + \mathsf{keysize}(\mathsf{DCF}_{n-f,1}) + \mathsf{keysize}(\mathsf{DCF}_{n,1}) + 8n - 2f + 2$. Concretely, for $n = 64, f = 24$ our key size is 5.7× and 1.9× lower than these 2 approaches, respectively. Our approach also has 2.9× and 1.9× lower AES evaluations in online phase.

6. Protocols for Softmax

To mirror the quantized training of [31], we calculate softmax accurately in floating-point. To do this securely, we make use of the state-of-the-art 2PC floating-point library, SecFloat [53]. 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 [53]. 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 6.1 and Section 6.2.

```
**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** $\hat{u}_b = p'_{b,3} \cdot (\hat{x} + 2^{n-f}) + p'_{b,2} \cdot \hat{x} + q_{b,\hat{j}} \mod N$
```

Figure 4: Protocol for $\mathsf{ReLUExt}$.

Floating-point representation. SecFloat [53] represents a 32-bit floating-point number α using four values $(z, s, e, m) \in \mathbb{FP}$ 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:

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

where $x_{\mathsf{max}} = \mathsf{max}(x_0, x_1, \ldots x_{d-1})$. 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 $(0, 1]$.

6.1. 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[3] holds:

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

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. 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:

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

with $|x|$ and k defined similarly as above. Compared to a natural approach of computing each sub-expression involving a comparison with a separate DCF key, we provide a novel protocol that uses only 2 DCF keys. Overall, for $n = 64$, we get a reduction of $1.8\times$ over naive approach. We provide a detailed description of our protocol, achieving the cost summarized below, in Appendix D.2.

Theorem 9. *$\Pi^{\mathsf{FixToFloat}}_{n,f}$ realizes $\mathsf{FixToFloat}_{n,f}$ securely such that $\mathsf{keysize}(\Pi^{\mathsf{FixToFloat}}_{n,f}) = \mathsf{keysize}(\mathsf{DCF}_{n,n}) + \mathsf{keysize}(\Pi^{\mathsf{select}}_n) + 2n^2 + 3n + 1$. In the online phase, the protocol requires $2n$ evaluations of $\mathsf{DCF}_{n,n}$ and communicates $4n + 2$ bits in 2 rounds.*

6.2. 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 $\Pi^{\mathsf{FloatToFix}}_{n,f}$ that realizes the above functionality $\mathsf{FloatToFix}_{n,f}$ for softmax outputs uses similar ideas as above and due to space constraints, we delegate details to the full version [34]. The following theorem summarizes its cost where $\Pi^{\mathsf{ARS}}_{n,f}$ is the protocol for arithmetic right shift of an n-bit value by f provided by [16].

Theorem 10. *$\Pi^{\mathsf{FloatToFix}}_{n,f}$ realizes $\mathsf{FloatToFix}_{n,f}$ securely such that $\mathsf{keysize}(\Pi^{\mathsf{FloatToFix}}_{n,f}) = \mathsf{keysize}(\mathsf{DCF}_{24,\mathbb{U}_2}) + \mathsf{keysize}(\Pi^{\mathsf{select}}_n) + \mathsf{keysize}(\Pi^{\mathsf{ARS}}_{n,f}) + 2049n + 35$. In the online phase, the protocol requires 1 evaluation of $\mathsf{DCF}_{24,\mathbb{U}_2}$ and $\Pi^{\mathsf{ARS}}_{n,f}$ and costs communication of $4n + 70$ bits in 3 rounds.*

6.3. End-to-end training

As discussed earlier in Section 5, 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 $\Pi^{\mathsf{stTrReLU}}_{n,f}$ (or $\Pi^{\mathsf{TRM}}_{n,f}$ 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 $\Pi^{\mathsf{FixToFloat}}_{n,f}$, call the protocol for softmax from SECFLOAT [53], and then finally convert

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

the secret-shared floating point numbers back to masked fixed-point numbers using $\Pi^{\mathsf{FloatToFix}}_{n,f}$. To compute the backward pass for ReLU and Maxpool, we re-use the comparison outputs from the forward pass and combine them with $\Pi^{\mathsf{select}}_{n}$ 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 E.

Preprocessing Cost. In the preprocessing phase, the dealer must both compute and transfer the keys to P_0 and P_1. Almost all the cost in computing the keys comes from computing the Beaver triples and DCF keys, and this can be approximated to be about $1 - 4\times$ the cost of running our protocol in the online phase. However, if the computed keys are to be transferred to P_0 and P_1 over a network, then this cost would dominate and depend on the network bandwidth. As an example, a single iteration of training of PIRANHA's VGG16 model on the CIFAR-10 dataset with batch size 128 would require transferring a key of size approximately 15.1 GB, which in our network setting of 9.4 Gbps would take about 12.8s. In comparison, the online phase of executing the same benchmark takes 2.5s. In a WAN setting with 293 Mbps bandwidth, transferring the keys would take 422s, while the online phase takes 38.4s.

7. Implementation

We implement ORCA as a C++ library for easy use. It contains optimized GPU kernels for the FSS gates described in Sections 4 and 5. 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 [59], 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, similar to PIRANHA [65], we stick to integer kernels to avoid the overheads of embedding.

Table 2 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 3, we described various optimization that we perform (AES, LAYOUT, MEM) to speed up DCF. As noted in the table, we apply several of the same GPU-centric optimizations to other kernels as suitable.

Some parts of our framework run on the CPU. As mentioned in Section 6, we use SECFLOAT [12] to compute softmax in floating point, which only has a CPU implementation. Furthermore, to go from fixed-point to floating point and back, we implement the protocols FixToFloat and FloatToFix on the CPU as these conversions have a tiny cost compared to floating-point softmax itself.

Func.	Description	Source	Optimizations
Matrix mult.	Multiplies two matrices	CUTLASS	–
Conv	Performs convolution	CUTLASS	–
DCF	The DCF described in [16]	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 2: Key components of accelerated FSS

Beyond the design optimizations mentioned in Section 3, 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 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 offline key generation with GPUs. We use cuRAND library [4] to generate the randomness on the GPU for the dealer, thus reducing the offline computation time.

We extended LLAMA [30], the state-of-the-art tool for FSS-based inference on CPUs, to support training by executing its inference protocols for fixed-point operations occurring in training and using ORCA's implementation for softmax. 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 8.3.

8. 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 input independent 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 in an online phase. We provide an empirical evaluation to justify the following claims:

- ORCA faithfully implements the quantized training algorithms in ML literature [31] that mix floating-point and fixed-point arithmetic (Figure 5). ORCA beats the accuracy of PIRANHA (the current state-of-the-art in accelerating secure training with GPUs) generated models in $4 - 22\times$ less time and with $43 - 98\times$ less communication (Table 3).

- On identical training and inference tasks, ORCA outperforms (GPU-based) PIRANHA by up to $10\times$ in latency and up to $14\times$ in communication (Table 4). ORCA also outperforms state-of-the-art CPU-based secure training [40] by up to $693\times$ (Table 6).
- ORCA's GPU-based protocols are up to $19\times$ more efficient than their CPU counterparts (Table 7). Moreover, the size of FSS keys required by ORCA's protocols is up to $3\times$ lower (see Table 8) than LLAMA, the state-of-the-art tool for FSS-based machine learning.
- ORCA enables sub-second ImageNet-scale inference of VGG-16 and ResNet-50 (Table 9) and outperforms state-of-the-art by an order of magnitude.

Parameter setting. We evaluate ORCA and PIRANHA on a fixed-point representation with bitwidth $n = 64$ and precision $f = 24$. The CPU-baselines [40] and CrypTen [41] also use the bitwidth $n = 64$.

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 [65] 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 from [40]: Model-B$_\mathsf{M}$ [40] and AlexNet$_\mathsf{C}$ [59]. For models from Gupta et al. [31], we use the 36k parameter CNN2$_\mathsf{M}$ model with 2 convolutions and the 200k parameter CNN3$_\mathsf{C}$ model with 3 convolutions.

Finally, we also evaluate secure inference for the ImageNet-1000 [25] dataset. The models here are much larger than the MNIST/CIFAR models as they operate 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 for ImageNet has $49\times$ more multiplications and $73\times$ more comparisons. We evaluate secure ImageNet inference with maxpools and stochastic truncations to match its floating-point accuracy. We omit secure ImageNet 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 46 GB of onboard memory and an AMD Epyc 7742 processor. Each machine sports nearly a TB of RAM and a Samsung 980 PRO NVMe SSD that supports about 6 GBps of sequential read bandwidth. We use four of the CPU cores for CPU-only experiments. We use four CPU cores in the GPU evaluations to support the parts of the computation that run on the CPU. Additionally, we use another CPU core to read keys from SSD to RAM.

	Accuracy		Time (in min)		Comm. (in GB)	
Dataset	PIRANHA	ORCA	PIRANHA	ORCA	PIRANHA	ORCA
MNIST	96.8 (-0.3%)	97.1	56 ($4.0\times$)	14	2,168.4 ($43.2\times$)	50.2
CIFAR-10	55 (-4.6%)	59.6	1170 ($22.5\times$)	52	65231.3 ($98.5\times$)	662.4
	55 (-14%)	69	1170 ($9.1\times$)	128	65231.3 ($39.4\times$)	1656.1

Table 3: ORCA matches accuracy of PIRANHA in lesser time and with lower communication on MNIST and CIFAR-10.

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 times reported for the baselines, for both training and inference do not include the time to load the pre-processing material into RAM. Similar to [65], for all the baselines and ORCA, we only report the online time[4].

8.1. End-to-end training with ORCA

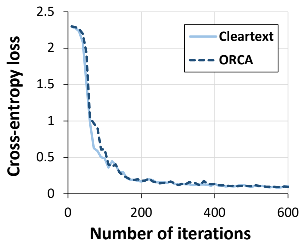
Our goal is to show that for a given classification task, ORCA can train a model that matches the accuracy of the models trained by PIRANHA, while incurring significantly less time and communication. Recall that PIRANHA's implementation of softmax leaks private information [63], while ORCA uses a secure floating-point softmax that helps provide end-to-end security.

On the MNIST dataset, PIRANHA reports an accuracy of 96.8% while training P-LeNet$_\mathsf{M}$, incurring about an hour in training time and about 2TB of communication. Table 3 shows that ORCA matches this accuracy with $4\times$ less time and $43\times$ less communication while training the CNN2$_\mathsf{M}$ model [31]. We evaluate ORCA on P-LeNet$_\mathsf{M}$ in Table 4.

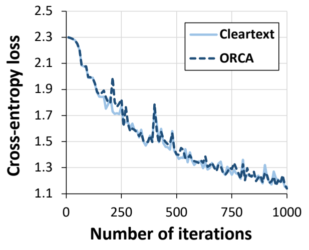
On the CIFAR dataset, which is a harder classification problem than MNIST, the improvements are even more pronounced. While training CNN3$_\mathsf{C}$ [31], ORCA outperforms PIRANHA's training of P-VGG16$_\mathsf{C}$ by 4.6% in accuracy, $22.5\times$ in latency, and $98\times$ in communication. We achieve the accuracy of 59.6% by training CNN3$_\mathsf{C}$ for 2 epochs. When training for 5 epochs, we achieve a much higher accuracy of 69% (14% better than PIRANHA) while still being $9.1\times$ faster and requiring $39\times$ lower communication than PIRANHA. We evaluate ORCA on P-VGG16$_\mathsf{C}$ in Table 4.

Accuracy. Since ORCA evaluates the models by Gupta et al. [31] faithfully, there are no accuracy gaps between (PyTorch) cleartext training and secure training. We show this empirically as well in Figure 5. Training these models

4. 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 8) from the machine running the dealer to the machines holding secret data.



(a) Training CNN2$_M$ with batch size 100 for 1 epoch.



(b) Training CNN3$_C$ with batch size 100 for 2 epochs.

Figure 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$_M$	training	166 (2.2×)	76	31 (5.7×)	5.45
	inference	57 (9.5×)	6	21 (8.2×)	2.56
P-LeNet$_M$	training	720 (4.1×)	174	474 (7.5×)	63
	inference	402 (8.9×)	45	335 (8.8×)	38
P-AlexNet$_C$	training	984 (5.1×)	193	606 (5.6×)	109
	inference	424 (7.2×)	59	324 (9.0×)	36
P-VGG16$_C$	training	18096 (7.3×)	2489	17083 (9.2×)	1854
	inference	14106 (10.2×)	1387	13589 (14.9×)	911

Table 4: Comparing against GPU baseline PIRANHA for one training iteration and inference with a batch size 128.

instead with PIRANHA approximations leads to 10% accuracy, i.e., that of a random classifier.

8.2. 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 8.2.1. In Section 8.2.2, we compare ORCA and state-of-the-art CPU-based secure training [40].

8.2.1. GPU baselines. We compare ORCA and PIRANHA in Table 4 on the benchmarks used by PIRANHA for both training and inference tasks. We observe that ORCA is up to 10× better in latency and 14× better in communication. Here, for a fair comparison, both ORCA and PIRANHA use average pools, linear approximations for softmax, and local truncations. Protocols in ORCA for these approximations are provided in the full version [34]. Since the communication of ORCA is much lower than PIRANHA, with lower bandwidth (e.g., WAN settings), these improvements are generally even higher (Table 5).

8.2.2. CPU baselines. The latest work in CPU-based secure training is by Keller and Sun [40], shown as KS in Table 6. Secure training in KS uses faithful maxpools and stochastic truncations, and does not support local truncations. To measure running time of KS, we instrument MP-SPDZ [38] to measure the time taken by one iteration of online phase after the preprocessing material has been loaded in RAM. Table 6 shows that ORCA (with stochastic truncations, maxpools and floating-point softmax) is up to 693× faster than KS, while incurring up to 261× less communication.

		Time (s)		Comm. (MB)	
Model	Task	PIRANHA	ORCA	PIRANHA	ORCA
P-SecureML$_M$	training	3.4 (2.6×)	1.3	31 (5.7×)	5.45
	inference	2.3 (8.2×)	0.28	21 (8.2×)	2.56
P-LeNet$_M$	training	12.3 (4.9×)	2.5	474 (7.5×)	63
	inference	8.6 (8.9×)	0.97	335 (8.8×)	38
P-AlexNet$_C$	training	19.9 (5.0×)	4	606 (5.6×)	109
	inference	12.2 (8.7×)	1.4	324 (9.0×)	36
P-VGG16$_C$	training	332.9 (8.7×)	38.4	17083 (9.2×)	1854
	inference	258.6 (13.6×)	19	13589 (14.9×)	911

Table 5: Comparing against GPU baseline PIRANHA for one training iteration and inference with a batch size 128 in a WAN setting with bandwidth 293 Mbits per second and ping time of 60ms.

	Time (in sec)		Comm. (in MB)	
Model	KS	ORCA	KS	ORCA
CNN2$_M$	10 (7.0×)	1.42	3102 (36.1×)	86
Model-B$_M$	17.2 (11.7×)	1.47	5796 (47.9×)	121
AlexNet$_C$	299.3 (148.9×)	2.01	46213 (104.3×)	443
CNN3$_C$	1685 (693.4×)	2.43	177152 (261.3×)	678

Table 6: Comparison against CPU baseline KS [40] for one training iteration with batch size of 100.

8.3. Improvements breakup

LLAMA [30] is the state-of-the-art in FSS-based protocols for ML tasks, and outperforms other tools such as AriaNN [57]. Conceptually, ORCA makes two performance improvements over LLAMA. First, it provides new protocols for operations occurring in ML tasks with smaller FSS keys and lower compute. Second, it accelerates these protocols with GPUs. In Table 7, we investigate the individual contributions of protocols and GPUs to the overall speedups in runtime over LLAMA. We compare concrete communication of LLAMA and ORCA in (Appendix A.2) and note that ORCA incurs slightly (up to 25%) higher communication.

Overall, the protocols of ORCA, described in Section 5 provide roughly 3× reduction in key size over LLAMA (see Table 8) and lower number of AES calls. As key read is not the performance bottleneck in CPU implementation, new protocols translate to $< 2.2\times$ improvement in running time of ORCA run on CPU over LLAMA in Table 7. GPU

Dataset	Model	Task	LLAMA	Our CPU	Our GPU
MNIST	CNN2$_\mathsf{M}$	training	3.54 (2.5×)	2.57 (1.8×)	1.42
		inference	1.57 (31.4×)	0.93 (18.6×)	0.05
CIFAR-10	CNN3$_\mathsf{C}$	training	38.58 (15.9×)	18.85 (7.8×)	2.43
		inference	31.89 (42.0×)	14.77 (19.4×)	0.76

Table 7: Comparison of latency (in seconds) of a single iteration of training and inference (batch size 100) between LLAMA, our CPU and GPU implementations.

Dataset	Model	Task	LLAMA	ORCA
MNIST	CNN2$_\mathsf{M}$	training	2.18 (2.9×)	0.75
		inference	1.54 (3.0×)	0.51
CIFAR-10	CNN3$_\mathsf{C}$	training	34.66 (3.0×)	11.51
		inference	29.27 (3.1×)	9.32

Table 8: Comparison of key size (in GB) against LLAMA for a single iteration of training and inference (batch 100).

acceleration of our protocols brings running time further down by up to 19×. Reading huge FSS keys becomes a performance bottleneck after accelerating compute with GPUs and key size reduction through our new protocols becomes crucial. For example, in CNN3$_\mathsf{C}$ training, with a key size of 11.51 GB (Table 8), ORCA with GPU runs in 2.43s (Table 7). Without ORCA's key size reduction, it would take 6.2s to move the 34.66 GB key (Table 8) from our SSD to RAM, which will degrade the overall latency from 2.43s to 6.2s. Even with smaller key size, overlapping key read and the rest of the computation is crucial to keep latency low (Table 12, Appendix A.4).

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 [53] 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, once keys are in RAM, out of 1.4s compute in 1 training iteration of CNN2$_\mathsf{M}$, 1.3s ($\approx 92\%$) is taken by softmax.

We show that GPU accelerates both linear and non-linear layers by an order of magnitude in Figure 6, Appendix A.1.

8.4. Secure ImageNet inference

Table 9 shows that ORCA enables sub-second secure ImageNet-1000 inference. We compare quantitatively against the state-of-the-art secure ImageNet inference works that use the same threat model as ORCA. Here, CrypTen [41] uses GPUs and LLAMA [30] is CPU only. We under-approximate CrypTen's runtime by dividing its reported communication with the network bandwidth of our setup.

The accuracy of secure inference with ORCA using bitwidth $n = 64$ matches the cleartext floating-point ac-

Model	LLAMA	CrypTen	ORCA	
			$n = 64$	time $[n, f]$
VGG-16	54.93 (103.6×)	13.19 (24.9×)	1.21 (2.3×)	0.53 [32, 12]
ResNet-50	45.83 (67.4×)	5.76 (8.5×)	0.93 (1.4×)	0.68 [37, 12]
ResNet-18	12.03 (85.9×)	2.97 (21.2×)	0.28 (2.0×)	0.14 [32, 10]

Table 9: Runtime (in seconds) for secure ImageNet inference with LLAMA (CPU), CrypTen (GPU), ORCA with bitwidth $n = 64$ and precision $f = 24$, and ORCA with smaller bitwidth and precision $[n, f]$ that is sufficient to maintain floating-point model accuracy.

curacy over a validation set of 50,000 ImageNet images. It is well known that smaller bitwidths can be sufficient to maintain inference accuracy [24], [55]. We run ORCA with smaller bitwidths to obtain a further $1.4 - 2.3\times$ speedup over ORCA with bitwidth 64, while maintaining accuracy. Since CrypTen doesn't support these smaller bitwidths, we keep the bitwidth of baselines as 64. We report ORCA's key size for these tasks in Table 11 (Appendix A.3). Overall, ORCA is $8.5 - 103\times$ faster than baselines.

Although ORCA's time for ResNet-50 is over 13× and 118× better than the latency reported by CryptGPU [59] and Cheetah [32] respectively, such comparisons are unfair to these baselines as their threat models are different.

9. Related Work

This work crucially builds on several existing techniques. Gupta et al. [31] provide quantized training algorithms and models that preserves accuracy, which became the cleartext functionalities for our secure implementations. Alternatively, it is possible to not quantize and run the full training algorithm in floating-point [52], however, this option has higher cost. Various works have shown that GPUs are effective at accelerating MPC protocols [41], [59], [65]. Since communication overheads cannot be reduced by GPU acceleration, it is natural to consider low communication protocols like FSS [17]. We use PIRANHA's [65] proposal for evaluating linear layers efficiently with GPUs. For ReLU, non-interactive protocols are known [16]. We trade-off rounds to decrease the key size for commonly occurring ML operations like ReLU and ReLU followed by a Maxpool.

Secure training is a rich area with various techniques. There are solutions that lack cryptographic security guarantees like Trusted Execution Environments [68] and Federated Learning [58]. There are complementary techniques like Differential Privacy (DP) [14], [62] 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 [15], [37], [48].

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 [57], [66].
3-party honest majority. Three non-colluding parties run a secure training protocol [42], [47], [50], [59], [61], [63].
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 [23], [40], [41], [69].

Prior works in secure inference that don't address secure training include [21], [22], [30], [32], [35], [36], [43], [44], [46], [54]–[56]. Works that accelerate non-FSS protocols with GPUs include [28], [33], [46], [49], [51], [59], [65]. In particular, GFORCE [49] accelerates secure MNIST/CIFAR inference with GPUs, but neither evaluates ImageNet inference nor training. 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 [24], [27], [38], [42], [50], [69].

10. 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 52 minutes and ImageNet-1000 inference runs in sub-second. We also identify concrete challenges for future work: integrating newer PCIe5 hardware 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] 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.

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

[16] 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.

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

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

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

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

[21] 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.

[22] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "Astra: High throughput 3pc over rings with application to secure prediction," in *CCSW*, 2019.

[23] H. Chen, M. Kim, I. Razenshteyn, D. Rotaru, Y. Song, and S. Wagh, "Maliciously secure matrix multiplication with applications to private deep learning," in *ASIACRYPT*, 2020.

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

[25] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *CVPR*, 2009.

[26] J. Doerner and A. Shelat, "Scaling ORAM for secure computation," in *CCS*, 2017.

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

[28] T. K. Frederiksen, T. P. Jakobsen, and J. B. Nielsen, "Faster maliciously secure two-party computation using the gpu," in *SCN*, 2014.

[29] 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.

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

[31] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision." in *ICML*, 2015.

[32] 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.

[33] N. Husted, S. Myers, A. Shelat, and P. Grubbs, "Gpu and cpu parallelization of honest-but-curious secure two-party computation," in *ACSAC*, 2013.

[34] N. Jawalkar, K. Gupta, A. Basu, N. Chandran, D. Gupta, and R. Sharma, "Orca: Fss-based secure training with gpus," Cryptology ePrint Archive, Paper 2023/206, 2023.

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

[36] G. Kaissis, A. Ziller, J. Passerat-Palmbach, T. Ryffel, D. Usynin, A. Trask, I. Lima, J. Mancuso, F. Jungmann, M.-M. Steinborn, A. Saleh, M. Makowski, D. Rueckert, and R. Braren, "End-to-end privacy preserving deep learning on multi-institutional medical imaging," *NATURE MACHINE INTELLIGENCE*, 2021.

[37] M. Kelkar, P. H. Le, M. Raykova, and K. Seth, "Secure poisson regression," in *USENIX Security Symposium*, 2022.

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

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

[40] ——, "Secure quantized training for deep learning," in *ICML*, 2022.

[41] 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.

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

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

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

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

[46] 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.

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

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

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

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

[51] 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.

[52] D. Rathee, A. Bhattacharya, D. Gupta, R. Sharma, and D. Song, "Secure floating-point training," in *USENIX Security Symposium*, 2023.

[53] 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.

[54] 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.

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

[56] 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.

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

[58] 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.

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

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

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

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

[63] 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.

[64] K. Wang, D. Fussell, and C. Lin, "Fast Fine-Grained Global Synchronization on GPUs," in *ASPLOS*, 2019.

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

[66] 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.

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

[68] P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana, "Plinius: Secure and persistent machine learning model training," in *DSN*, 2021.

[69] 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

A.1. Microbenchmarks

We show acceleration of GPUs for linear and non-linear layers separately in Figure 6. Across microbenchmarks of varying sizes, GPU-based protocols in ORCA are an order of magnitude faster than their CPU counterparts.

A.2. Online communication

We show that ORCA's online communication is only slightly higher than LLAMA in Table 10.

Dataset	Model	Task	LLAMA	ORCA
MNIST	CNN2$_M$	training	76 (0.88×)	86
		inference	22 (0.79×)	28
CIFAR-10	CNN3$_C$	training	579 (0.85×)	678
		inference	365 (0.89×)	411

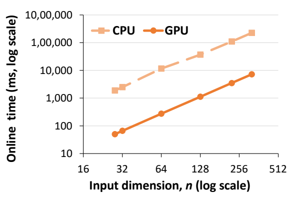
Table 10: Comparison of communication (in MB) of a single iteration of training and inference between LLAMA and ORCA. Batch size of 100 is being used.

A.3. Key size of ImageNet Inference tasks

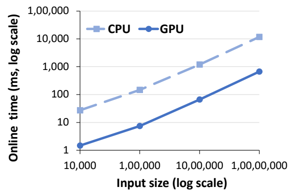
Table 11 shows the key size of various ImageNet inference tasks required by ORCA and LLAMA.

A.4. Overlapping key read

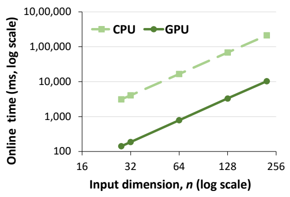
Table 12 shows the time ORCA spends on reading keys and on compute (including time to communicate) for one training iteration. Since key read and compute overlap in ORCA, a training iteration takes the larger of the two times.



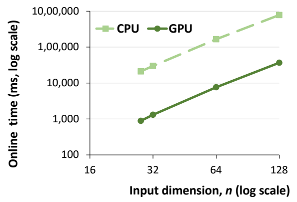
(a) Convolution on an $n \times n \times 64$ input with a 5×5 kernel, 64 output channels, stride 1, and padding 1. $n = 28, 32, 64, 128, 224, 320$.



(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. $n = 28, 32, 64, 128$.

Figure 6: Comparison of online time on CPU and GPU

Model	LLAMA	ORCA	
		$n = 64$	key size $[n, f]$
VGG-16	50.54	13.13	5.61 [32, 12]
ResNet-50	47.69	9.9	5.49 [37, 12]
ResNet-18	11.43	2.78	1.21 [32, 10]

Table 11: Key size (in GB) of secure ImageNet inference in LLAMA (CPU) with bitwidth $n = 64$, ORCA with bitwidth $n = 64$ and precision $f = 24$, and ORCA with smaller bitwidth and precision $[n, f]$ that is sufficient to maintain floating-point model accuracy.

Model	Key size (in GB)	Key read (ms)	Compute (ms)	ORCA (ms)
P-SecureML$_M$	0.03	5	**75**	76
P-LeNet$_M$	0.36	63	**172**	174
CNN2$_M$	0.75	129	**1407**	1415
Model-B$_M$	1.27	219	**1460**	1466
P-AlexNet$_C$	0.34	64	**192**	193
AlexNet$_C$	8.34	1387	**2011**	2011
CNN3$_C$	11.51	1911	**2424**	2432
P-VGG16$_C$	15.1	**2489**	2310	2489

Table 12: Key size, time of key read, and compute (inclusive of communication) for ORCA for one training iteration of different models. The last column reports the larger of key read time and compute time.

Appendix B. 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 ReLU and Maxpool are applied. As a result, many existing works carry out ReLU after MaxPool [43], [55], as this reduces the number of elements on which ReLU is calculated.

LLAMA [30] provides a protocol for MaxPool, which internally uses the spline-based protocol for ReLU from [16] 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) Stochastic 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 [30]. The theorem summarizes the cost for MaxPool over k elements.

Theorem 11. *There exists a protocol $\Pi^{\mathsf{TRM}}_{n,f,k}$ which realizes stochastic-truncation + ReLU + MaxPool of k elements securely such that $\mathsf{keysize}(\Pi^{\mathsf{TRM}}_{n,f,k}) = k \cdot \mathsf{keysize}(\hat{\Pi}^{\mathsf{stTR}}_{n,f}) + (k - 1) \cdot \mathsf{keysize}(\hat{\Pi}^{\mathsf{ReLU}}_{n-f}) + \mathsf{keysize}(\Pi^{\mathsf{ReLUExt}}_{n-f,n})$. Its online phase requires k evaluations of $\hat{\Pi}^{\mathsf{stTR}}_{n,f}$, $k - 1$ evaluations of $\hat{\Pi}^{\mathsf{ReLU}}_{n-f}$, 1 evaluation of $\Pi^{\mathsf{ReLUExt}}_{n-f,n}$ and communication of $2(2k - 1)(n - f + 1) + 8$ bits in $2k + 1$ rounds.*

In contrast, the baseline [30] has keysize roughly $k \cdot (\mathsf{keysize}(\mathsf{DCF}_{n,2n}) + 5n) + \mathsf{keysize}(\mathsf{DCF}_{n-1,2n}) + n$. 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 $2.5\times$.

Appendix C. Stochastic Truncate-Reduce

C.1. Protocol for Stochastic Truncate Reduce

We provide a formal protocol in Figure 7.

C.2. Security Proof

For ease of proof, we consider a real interaction where along with r^{in} and $\mathsf{r}^{\mathsf{out}}$, $\mathsf{Gen}^{\mathsf{stTR}}_{n,f}$ is paramterized by $s \in \mathbb{U}_{2^f}$, where s is the associated randomness of stochastic truncate-reduce. Similarly, we consider an ideal world that is also parameterized by $\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}, s$. In real interaction, P_0, P_1 run $\mathsf{Eval}^{\mathsf{stTR}}_{n,f}$. In the ideal interaction, P_0, P_1 send $\hat{x}$ to ideal

```
Stochastic Truncate-Reduce $\Pi^{\mathsf{stTR}}_{n,f}$
$\mathsf{Gen}^{\mathsf{stTR}}_{n,f}(\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$ :
  1: $s \xleftarrow{\$} \mathbb{U}_{2^f}$
  2: $r^{(t)} = \mathsf{r}^{\mathsf{in}} \mod 2^f$
  3: $\hat{s} = s + r^{(t)} \mod 2^f$
  4: $(k_0^{>}, k_1^{>}) \leftarrow \mathsf{Gen}^{>}_f(1^\lambda, \hat{s}, 1, \mathbb{U}_2)$
  5: $z \xleftarrow{\$} \mathbb{U}_2$; $r^{(w)} = \mathsf{extend}(z, n-f)$
  6: $r = \mathsf{r}^{\mathsf{out}} - \mathsf{TR}(\mathsf{r}^{\mathsf{in}}, f) - 1\{\hat{s} < r^{(t)}\}$
  7: share $r, r^{(w)}$
  8: For $b \in \{0,1\}$, $k_b = k_b^{>} || r_b^{(w)} || r_b$

$\mathsf{Eval}^{\mathsf{stTR}}_{n,f}(b, k_b, \hat{x})$ :
  1: Parse $k_b$ as $k_b^{>} || r_b^{(w)} || r_b$
  2: $\hat{t} = \hat{x} \mod 2^f$
  3: $\hat{w}_b \leftarrow \mathsf{Eval}^{>}_f(b, k_b^{>}, \hat{t}) + r_b^{(w)} \mod 2$
  4: $\hat{w} = \mathsf{reconstruct}(\hat{w}_b)$
  5: $c_b = r_b^{(w)} + \mathsf{extend}(\hat{w}, n-f) \cdot (b - 2r_b^{(w)})$
  6: return $\hat{y}_b = b \cdot \mathsf{TR}(\hat{x}, f) + r_b + c_b$
```

Figure 7: Protocol for stTR.

functionality, that unmasks it and computes the output. That is, computes $\hat{y} = \mathsf{TR}(\hat{x} - \mathsf{r}^{\mathsf{in}}, f) + 1\{s < x_0\} + \mathsf{r}^{\mathsf{out}}$, and sends shares of $\hat{y}$ to P_0, P_1. Below, we describe the simulator that simulates the view of a real world adversary in such a real interaction.

Simulator. For $b \in \{0,1\}$ the simulator $\mathsf{Sim}^{\mathsf{stTR}}_b$ for stochastic truncate reduce is given $\hat{x}$ and $\hat{y}_b$ and has to simuate the view of party b, i.e., messages $k_b^{>} || r_b^{(w)} || r_b$ from the dealer and $\hat{w}_{1-b}$ from P_{1-b}. The simulator follows these steps:

1) Pick $r^{(w)}_{b,\mathsf{sim}}$ at random from $\mathbb{U}_{2^{n-f}}$.
2) Set $\hat{t} = \hat{x} \mod 2^f$.
3) Invoke $\mathsf{Sim}^{<}_b$ to get $k^{<}_{b,\mathsf{sim}}$.
4) Set $\hat{w}_{b,\mathsf{sim}} \leftarrow \mathsf{Eval}^{<}_f(b, k^{<}_{b,\mathsf{sim}}, \hat{t}) + r^{(w)}_{b,\mathsf{sim}}$.
5) Pick $\hat{w}_{\mathsf{sim}} \in \mathbb{U}_2$ at random and set $\hat{w}_{1-b,\mathsf{sim}} = \hat{w}_{\mathsf{sim}} \oplus \hat{w}_{b,\mathsf{sim}}$.
6) Run step 5 to compute $c_{b,\mathsf{sim}}$.
7) Set $r_{b,\mathsf{sim}} = \hat{y}_b - b \cdot \mathsf{TR}(\hat{x}, f) - c_{b,\mathsf{sim}}$.

It is easy to argue that the joint distribution of the view of the adversary and output of honest party in the real interaction is indistinguishable from that in the ideal interaction in the presence of the above simulator.

Appendix D. Interconversion between Fixed and Float

We describe our protocol for fixed-to-float here and defer the description of float-to-fixed to full version [34]. Our fixed-to-float protocol uses an FSS gate for multiple interval containment from [16] summarized below.

D.1. 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: $y_i = 1$ if $p_i \leqslant x \leqslant q_i$ and 0 otherwise. [16] provides a protocol $\Pi^{\mathsf{MIC}}_{n,m,\boldsymbol{p},\boldsymbol{q}}$ for $\mathsf{MIC}_{n,m,\boldsymbol{p},\boldsymbol{q}}$. We 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 12 (Multiple Interval Containment [16]). *$\Pi^{\mathsf{MIC}}_{n,m,\boldsymbol{p},\boldsymbol{q}}$ realizes $\mathsf{MIC}_{n,m,\boldsymbol{p},\boldsymbol{q}}$ securely such that $\mathsf{keysize}(\Pi^{\mathsf{MIC}}_{n,m,\boldsymbol{p},\boldsymbol{q}}) = \mathsf{keysize}(\mathsf{DCF}_{n,n}) + mn$. In the online phase, m evaluations of $\mathsf{DCF}_{n,n}$ are needed.*

D.2. Protocol for $\Pi^{\mathsf{FixToFloat}}_{n,f}$

Recall that we need to compute z, s, e, m using the expressions in Section 6.1. A natural way to compute $\mathsf{FixToFloat}_{n,f}$ securely is by using the existing protocols for Zero-Test [19], 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 $n(4\lambda + 3n + 15) + 4\lambda + 3$ bits. We describe an alternative approach to achieve the same result using two DCF keys and a key size of $n(2\lambda + 3n) + 14n - 30\lambda + 209$ 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 D.1) to get shares of a one-hot vector that specifies which interval x belongs to. We can then 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. For truncate-reduce, we use a protocol $\Pi^{\mathsf{TR}}_{n,f}$ similar to the protocol for stochastic truncate-reduce (Section 5.2.1) with same costs.

We describe the protocol for $\mathsf{FixToFloat}_{n,f}$ in Figure 8 and provide its security proof in the full version [34].

Appendix E. 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

```
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: $r^{(z)} \xleftarrow{\$} \mathbb{U}_N$
9: $c = r^{(u)} \cdot r^{(y)} + r^{(z)}$
10: $(k_0^{\mathsf{TR}}, k_1^{\mathsf{TR}}) \leftarrow \mathsf{Gen}^{\mathsf{TR}}_{n,n-24}(r^{(z)}, 0)$
11: share $(r^{(s)}, r^{(u)}, r^{(y)}, c)$
12: $\forall b \in \{0,1\}, k_b = k_b^{(\mathsf{MIC})} || k_b^{(\mathsf{select})} || r_b^{(s)} || r_b^{(u)} || r_b^{(y)} || c_b || k_b^{\mathsf{TR}}$

$\mathsf{Eval}^{\mathsf{FixToFloat}}_{n,f}(b, k_b, \hat{x})$ :
1: Let $\boldsymbol{p} = \boldsymbol{p}^{(n)}$ and $\boldsymbol{q} = \boldsymbol{q}^{(n)}$
2: Parse $k_b$ as $k_b^{(\mathsf{MIC})} || k_b^{(\mathsf{select})} || r_b^{(s)} || r_b^{(u)} || r_b^{(y)} || c_b || k_b^{\mathsf{TR}}$
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: $\hat{z}_b = b \cdot \hat{y} \cdot \hat{u} - r_b^{(y)} \cdot \hat{u} - r_b^{(u)} \cdot \hat{y} + c_b$
13: $\hat{z} =$ reconstruct $(\hat{z}_b)$
14: $m_b \leftarrow \mathsf{Eval}^{\mathsf{TR}}_{n,n-24}(b, k_b^{\mathsf{TR}}, \hat{z})$
15: return $(z_b, s_b, e_b, m_b)$
```

Figure 8: Protocol for FixToFloat.

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 locally subtracted from the output to calculate the 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 Π^{select}_n, while for MaxPool, the protocol for bitwise-AND and Π^{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 $\Pi^{\mathsf{stTr}}_{n,f}$.

Appendix F. Security proof of ReLU-Extend

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

1) Generates $k^{<}_{b,\mathsf{sim}}$ by invoking $\mathsf{Sim}^{<}_b$. Runs $\mathsf{Eval}^{<}_{n-f}(b, k^{<}_{b,\mathsf{sim}}, \hat{x})$ to obtain $w_{b,\mathsf{sim}}$.
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 $\hat{w}_{b,\mathsf{sim}}, \hat{d}_{b,\mathsf{sim}}$ using steps $2, 4, 5$ in $\mathsf{Eval}^{\mathsf{ReLUExt}}_{n-f,n}$.
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 $\boldsymbol{p}_{b,\mathsf{sim}} \in \mathbb{U}_N^4$ and sets $\boldsymbol{p}'_{b,\mathsf{sim}} = \boldsymbol{p}'_{b,\mathsf{sim}} \ggg i_{\mathsf{sim}}$.
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 $\boldsymbol{q}_{b,\mathsf{sim}} = \{q_{b,\mathsf{sim},0}, q_{b,\mathsf{sim},1}\}$

Appendix G. Meta-Review

G.1. Summary

This paper describes methods to speed up secure machine learning (ML) training for two-party (2PC) scenarios using Function Secret Sharing (FSS) and GPU computations. To accelerate FSS, the paper proposes several optimizations/engineering steps that are tailored to the design of GPU's inner workings. Additionally, the system presents variations of FSS primitives that are designed to optimize the observed performance bottlenecks on naive implementations.

G.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

G.3. Reasons for Acceptance

1) Creates a new tool to enable future science: The proposed system moves towards practical secure inference and training by accelerating FSS-based protocols through both systems' advances with GPUs and new cryptographic techniques.
2) Provides a valuable step forward in an established field: The proposed system outperforms the competition in all tested performance and accuracy metrics.
3) Establishes a new research direction: The authors develop practical optimizations based on their preliminary analysis that identified the bottlenecks when using GPU in the context of FSS. The reviewers acknowledged the authors' commendable efforts in integrating advanced techniques, including function secret sharing (FSS), fixed-point and floating-point arithmetic, and GPU acceleration, to develop the comprehensive system proposed in this study.

G.4. Noteworthy Concerns

1) The reviewers mentioned that including some details about the 2PC preprocessing in the main text would be beneficial.
2) The reviewers agreed that the protocols' writing style is dense, making it difficult to understand the key contributions of each protocol. The paper would benefit if the presentation style of the corresponding sections became more broadly accessible.