SIRNN: A Math Library for Secure RNN Inference

Deevashwer Rathee*
Microsoft Research
deevashwer@berkeley.edu

Mayank Rathee*
Microsoft Research
mayankr@berkeley.edu

Rahul Kranti Kiran Goli
Microsoft Research
t-grahulk@microsoft.com

Divya Gupta
Microsoft Research
divya.gupta@microsoft.com

Rahul Sharma
Microsoft Research
rahsha@microsoft.com

Nishanth Chandran
Microsoft Research
nichandr@microsoft.com

Aseem Rastogi
Microsoft Research
aseemr@microsoft.com

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00086

***Abstract*— Complex machine learning (ML) inference algorithms like recurrent neural networks (RNNs) use standard functions from math libraries like exponentiation, sigmoid, tanh, and reciprocal of square root. Although prior work on secure 2-party inference provides specialized protocols for convolutional neural networks (CNNs), existing secure implementations of these math operators rely on generic 2-party computation (2PC) protocols that suffer from high communication. We provide new specialized 2PC protocols for math functions that crucially rely on lookup-tables and mixed-bitwidths to address this performance overhead; our protocols for math functions communicate up to $423\times$ less data than prior work. Furthermore, our math implementations are numerically precise, which ensures that the secure implementations preserve model accuracy of cleartext. We build on top of our novel protocols to build SIRNN, a library for end-to-end secure 2-party DNN inference, that provides the first secure implementations of an RNN operating on time series sensor data, an RNN operating on speech data, and a state-of-the-art ML architecture that combines CNNs and RNNs for identifying all heads present in images. Our evaluation shows that SIRNN achieves up to three orders of magnitude of performance improvement when compared to inference of these models using an existing state-of-the-art 2PC framework.**

***Index Terms*—privacy-preserving machine learning; secure two-party computation; recurrent neural networks; math functions; mixed-bitwidths; secure inference**

I. INTRODUCTION

In the problem of secure inference, there are two parties: a server that holds a proprietary machine learning (ML) model and a client that holds a private input. The goal is for the client to learn the prediction that the model provides on the input, with the server learning nothing about the client's input and the client learning nothing about the server's model beyond what can be deduced from the prediction itself. Theoretically, this problem can be solved by generic secure 2-party computation (2PC) [49], [115]. Recently, this area has made great strides with the works of [5], [10], [17]–[20], [25], [27], [32], [35], [37], [39], [47], [58], [64], [69], [73], [83], [90]–[92], [99]–[102], [110] that have made it possible to run secure inference on deep neural networks (DNNs). Frameworks for secure inference like nGraph-HE [18], [19], MP2ML [17], CrypTFlow [73], [99], and SecureQ8 [37] go one step further and can automatically compile models trained in TensorFlow/PyTorch/ONNX to 2-party or 3-party computation protocols secure against semi-honest adversaries.

While such systems cover the secure inference of some famous Convolutional Neural Networks (CNNs) (e.g. ResNet [56], DenseNet [61] and MobileNet [105]) that exclusively use simple non-linear functions such as ReLU and Maxpool, other important architectures such as Recurrent Neural Networks (RNNs) or architectures that combine RNNs and CNNs [104] use math functions, such as exponentiation, reciprocal square root, sigmoid and tanh, extensively. These RNN-based architectures are the models of choice when dealing with sequential or time series data like speech [36], [59], [112]. Hence, for widespread adoption of secure inference, especially in the RNN application domains, a robust support for math functions is of paramount importance.

We focus on 2-party inference secure against semi-honest adversaries[1]. In this setting, works that implement math functions fall into three categories. First, works that develop general purpose math libraries [9], [66] using high-degree polynomials. Second, works that use boolean circuits to implement math functions [102]. Third, works that use ad hoc piecewise linear approximations [83] that require developer intervention for each dataset and each model to balance accuracy and latency, an unacceptable ask in the context of automated frameworks for secure inference. All of these three approaches rely on 2PC protocols from [41], [66], [115] and suffer from huge performance overheads.

In this work, we design math functionalities that are both provably precise and efficiently realizable via novel 2PC protocols that we have developed. The performance of all 2PC implementations depend critically on the *bitwidth*. While prior works use a uniform bitwidth for the whole inference, our math functionalities use non-uniform (or mixed) bitwidths: they operate in low bitwidths and go to high bitwidths only when necessary. Hence, we have developed new protocols that enable switching between bitwidths and operating on values of differing bitwidths. Our 2PC protocols for math functionalities have upto $423\times$ lower communication than prior works (Section VI-A). We have implemented these in SIRNN[2], a

* The first two authors have equal contribution.

[1] We relegate comparisons with works that need additional parties for security, e.g., 3-party computation (3PC) to Section VII.

[2] Read as "siren", SIRNN stands for Secure Inference for RNNs.

DOI 10.1109/SP40001.2021.00086

library for end-to-end DNN inference, and evaluated on RNN-based models. While we focus on math functions occuring in RNNs, our recipe for designing math functionalities is general and can be used in other contexts. Furthermore, our math functionalities and non-uniform bitwidth protocols can also be used in non-RNN contexts and are of independent interest.

A. Results in detail

New approximations for math functions. In this paper, we provide *provably precise* functionalities, i.e. cleartext implementations, for exponentiation, sigmoid, tanh, and reciprocal of square root, that have been designed to minimize cryptographic overheads. Exponentiation is used in RBF kernels [55], sigmoid and tanh in RNNs with LSTM [59] and GRU [36] cells, and reciprocal square root in `L2Normalization`, where a vector u is scaled down to a unit vector by multiplying each entry of u by $\frac{1}{\sqrt{u^T u}}$. In a sharp departure from prior work in 2PC, our functionalities follow the well-known paradigm of using lookup tables (LUT) to get a good initial approximation of the math function followed by an iterative algorithm such as Goldschmidt's iterations [50] to improve upon this approximation. We take inspiration from embedded systems [51], [63], [72], [113] where the goal of minimizing memory consumption has led to efficient low-bitwidth implementations based on fixed-point arithmetic. Our functionalities manipulate variables with different bitwidths to maintain precision while using minimal bitwidths. Furthermore, we formally verify that our functionalities provide precision guarantees similar to those provided by standard math libraries (Section V-D).

Novel 2PC Protocols. We provide efficient protocols for bitwidth switching (both extensions and truncations) and operating on values with differing bitwidths so that our secure implementations mimic the behavior of the cleartext math functionalities that operate on non-uniform minimal bitwidths. As a baseline, another option is to use existing 2PC protocols that work with a uniform bitwidth (for all values) that is large enough to accommodate all intermediate values, i.e., avoids integer overflows. Similar to prior works, this would force us to work over much larger rings such as $\mathbb{Z}_{2^{64}}$. Since the complexity of secure protocols grows proportionally with the bitwidth used, our use of non-uniform bitwidth leads to much more communication efficient protocols than the naïve approach of uniform bitwidth. We consider 4 main building blocks to achieve this: (a) Extension - to increase bitwidths, (b) Truncation - to decrease bitwidths (and precision), (c) Multiplication - to multiply an m and n bit integer into an $(m+n)$-bit output to avoid overflows (this product is later truncated to have the right bitwidth required for further operations), and (d) Digit decomposition - to extract relevant substrings (that we call *digits*) of the input bitstring using which table lookups are performed. Moreover, the fixed-point cleartext code of our benchmarks also uses non-uniform bitwidths in linear layers such as matrix multiplications and convolutions, and we use our protocols for efficient realizations of the same.

Secure Inference Library. We have implemented our protocols for math functions in a new library, called SIRNN[3], for DNN inference. We evaluate SIRNN on three state-of-the-art models that use fixed-point arithmetic with non-uniform bitwidths [72]. Two of the models, one for the standard Google-30 dataset and the other for sports training, use an RNN architecture that provides accurate analysis of time series data [74]. For the Google-30 dataset, the task is to recognize commands like "Yes" and "No" from speech data, whereas the sports training model provides performance feedback to a sportsperson from sensor readings. To the best of our knowledge, this is the first empirical evaluation of secure inference of RNNs on time series inputs like speech and sensor readings. While it is possible to perform this inference using generic 2PC protocols, the overheads are intractable. To evaluate this quantitatively, we implemented our benchmarks using the state-of-the-art ABY [41] framework and this baseline is three orders of magnitude worse in latency and communication.

Our third model uses an architecture that combines RNNs and CNNs for the task of finding human heads in images [104]. This model uses the reciprocal square root function that is not supported by any of the prior works on secure inference. Additionally, it makes roughly 3 million calls to sigmoid and tanh each. In contrast, prior works on secure inference evaluated on models with less than 3000 calls to sigmoid/tanh [83], [102]. SIRNN can run the Heads model securely in under 7 minutes. To summarize, we make three key contributions:

1) We provide cryptographically friendly new approximations to math functions exponential, sigmoid, tanh and reciprocal square root that are provably precise (Section V).
2) We provide novel 2PC protocols for non-uniform bitwidths (Section IV) that realize these math functionalities efficiently (up to $423\times$ lower communication than prior work, Section VI-A).
3) We implement these secure implementations in the library SIRNN that provides the first secure inference of RNNs on speech and time series sensor data and a model that combines RNNs and CNNs. SIRNN outperforms state-of-the-art by three orders of magnitude in size of benchmarks (given by number of calls to math functions), latency and communication (Section VI-C). Furthermore, because of the high numerical precision of our math implementations, SIRNN has no loss in model accuracy.

The rest of the paper is organized as follows. We first provide a motivating example and an overview of our technical results in Section II. After discussing the necessary background in Section III, we provide our novel protocols in Section IV. The math functionalities are discussed in Section V with their formal verification in Section V-D. Section VI provides our evaluation on microbenchmarks, i.e., math functions in isolation (Table I & Table II), DNNs used by prior work that use math functions (Table III), and our RNN-based

[3]Implementation is available at https://github.com/mpc-msri/EzPC.

```
int16[2][2] W = ... ; int16[2] x = ...;
int16[2][2] U; int32[2] V; int32[2][2] T; int32[2] S;
U[0][0] = W[0][0]-x[0];  U[0][1] = W[0][1]-x[1]; , ...
T[0][0] = U[0][0]*U[0][0], ...
V[0] =  ((T[0][0] >> 12) + (T[0][1] >> 12), ...
S[0] = exp(-V[0], 32, 12, 32, 30), ...
return sign(S[0] - S[1])
```

Fig. 1: Fixed-point code for SVM with RBF kernel

benchmarks (Table IV). Finally, we discuss other related work in Section VII.

II. Overview

We now present an overview of our approximations for math functions and the building block protocols required to realize them. We begin with a motivating example of an inference task that crucially uses math functions; this will help us highlight concepts such as scale and bitwidth changes.

Motivating example. Support vector machines (SVMs) are one of the most widely used classical ML algorithms. While prior work on secure inference has used SVMs with polynomial kernels [76], [80], [87], [98] (that helps SVMs perform classification in exponentially large dimensions), the more powerful and hence widely used Radial Basis Function (RBF) kernels (that operate on infinite dimensions) [55] crucially relies on computing exponentiations, i.e., $e^x, x < 0$. No prior work on secure 2PC inference supports RBF.

Consider the simple task of predicting rain using a feature vector $x \in \mathbb{R}^2$, where $x[0]$ and $x[1]$ are temperature and humidity respectively, and the output is yes ($y = -1$) or no ($y = 1$). An SVM with RBF model infers the result using

$$\text{sign}\left(\sum_{i=1}^{k} c_i e^{-\gamma^2||W_i - x||^2}\right)$$

where the vectors $W_i \in \mathbb{R}^2$ are part of the model and $c_i \in \{-1, 1\}$. Here, $||W_i - x||^2$ is the square of the L2 norm or the Euclidean distance between W_i and x. Let $k = 2$, $\gamma = 1$, $c_0 = 1$ and $c_1 = -1$.

Scales and bitwidths. Since 2PC is much more efficient over integers than floating-point [29], [73], automated float-to-fixed converters [14], [24], [51], [72], [89], [94] can be used to express this model as computation over integers using fixed-point arithmetic. In fixed-point arithmetic, $r \in \mathbb{R}$ is (approximately) represented using an ℓ-bit integer $\lfloor r \cdot 2^s \rfloor \bmod 2^\ell$, where ℓ is the *bitwidth* and $s \in \mathbb{Z}$ is the *scale*. Hence, fixed-point integer a with scale s denotes $\frac{a}{2^s} \in \mathbb{R}$.

Consider the fixed-point code for our example given in Figure 1 generated by a float-to-fixed converter. The code stores the input x and the model parameters W as 16-bit integers with scale 12 (scale 12 is a common setting used in several prior works on secure inference [73], [92], [99]). To compute the inference result, it first computes $U_i = W_i - x$ where U has scale 12 using standard integer subtraction. Next, it computes $T = U \odot U$, where $\odot$ is pointwise multiplication. Since U has 16-bit entries, to avoid integer overflows, the entries of T must be 32-bits wide. Standard integer multiplication accumulates the scale and hence entries in T have a scale of 24. Thus, the code right shifts the entries of T by 12 to bring the scale back to 12 and accumulate them in $V_i = ||W_i - x||^2$. Next, it calls exponentiation on *negative* inputs of bitwidth 32 and scale 12 and produces the result S with bitwidth 32 and scale 30. The final result is the sign of $c_0 S[0] + c_1 S[1]$. SIRNN incurs less than 30KB of communication to run this code.

Observe that the fixed-point code in Figure 1 frequently changes bitwidths and scales with each operation. As we describe in Figure 3 (Section V), our math functionality for exponential would require multiplying two 32-bit values to compute an intermediate 64-bit result. Now, if we had to implement Figure 1 using existing 2PC protocols, we would be forced to use uniform bitwidth of at least 64 for all variables. In particular, the bitwidths of x, W, U, T, V, S will all be 64 instead of 16 or 32. More generally, the requirement of a high bitwidth even in one operation, coupled with the requirement of uniform bitwidths, raises the bitwidths of all variables and operations throughout an inference task, resulting in a communication blowup. In contrast SIRNN provides novel protocols for these low-level operations of switching bitwidth and scale and multiplying values of small bitwidth into large bitwidth. Ensuring that bitwidths used in secure code mimic the bitwidths used in low-bitwidth cleartext code, is the key factor in low communication complexity of our secure math functions.

Next we give an overview of our approximations for math functions followed by building blocks for our protocols.

A. Our approximations for math functions

Our math functionalities are designed keeping cryptographic costs in mind. We first use lookup tables (LUT) to get a good initial approximation of the math functions and then run an iterative algorithm such as Goldschmidt's iterations to improve upon this approximation. Larger LUTs lead to more precise results. However, the communication of secure protocol for LUTs grows linearly with size of LUT. Hence, we need to strike a balance to obtain implementations that are both precise and communication efficient. Thus, for exponentiation for negative inputs, we break the input bitstring x into smaller d-length substrings (via digit decomposition) that are used to index multiple 2^d-sized LUTs. The looked up values are multiplied into high bit intermediate results which are then truncated to match the specified output bitwidth and scale.

Sigmoid and tanh reduce to exponentiating negative values and reciprocating values between 1 and 2. For the latter, Ito *et al.* [63] provide a method for initial approximation of reciprocal using an LUT. After obtaining an initial approximation with ℓ bit entries and $\ell - 2$ bits of fractional part, we iterate using standard Goldschmidt's method. To make these iterations communication efficient, we run them using fixed-point arithmetic with non-uniform bit-widths. Our implementation for reciprocal square root is similar but requires additional

work to shift the initial input to be between 1 and 2 using the most significant non-zero bit (MSNZB).

B. 2PC protocols in SIRNN

The 2PC protocols in SIRNN are based on 4 building blocks: (a) Extension; (b) Truncation; (c) Multiplication; and (d) Digit decomposition. Our protocols mimic the low bitwidths used by cleartext fixed-point code, and work over power-of-2 rings, i.e. $\mathbb{Z}_{2^\ell}$. Let $\lambda = 128$ be the computational security parameter.

a) Extension: This is used to lift values from smaller ring $\mathbb{Z}_{2^m}$ to larger ring $\mathbb{Z}_{2^n}$ (i.e. $m < n$). Although extension has been considered in honest majority three-party computation [67], there are no specialized 2PC protocols for it. A natural baseline, however, is provided by Yao's garbled circuits[4] (GC) [115], which requires around $\lambda(4m + 2n)$ bits of communication to reconstruct and re-share. In contrast, our protocol requires around λm bits of communication, that is roughly $6\times$ better than GC.

b) Truncation: This operation is used to reduce scale and is often used after multiplication. We require 4 kinds of truncation operations for ℓ-bit values by s bits: logical and arithmetic right shifts (that preserve the bitwidth), truncate-and-reduce (outputs the truncated value in $\mathbb{Z}_{2^{\ell-s}}$), and division by 2^s. State-of-the-art protocol for arithmetic right shift (ARS) was given by [99] with communication roughly $\lambda(\ell + s)$ that can also be used for logical right shift and truncate-and-reduce. We give a new protocol for logical/arithmetic right shift with communication $\approx \lambda\ell$, i.e., independent of λs. Moreover, most of our math functionalities require only truncate-and-reduce that decreases both scale and bitwidth. We show how to achieve this in only $\approx \lambda(s+1)$ bits of communication. Finally, our fixed-point benchmarks also require a division by power-of-2 operation that is different from ARS for negative x and outputs $\lceil x/2^s \rceil$. Our protocol for this division requires roughly $4.5\times$ less communication than GC.

c) Multiplication: We consider the functionality for multiplying an m-bit integer with an n-bit integer to produce an $\ell = (m + n)$-bit output. This choice of ℓ ensures that there are no overflows. A similar functionality has been considered in the 3-party setting [67] that extends both operands to ℓ bits and then invokes existing multiplication protocols over ℓ bits. This approach can be used in 2PC setting as well using our optimized protocols for extension (that are $6\times$ better than GC). We provide an alternate protocol that requires $1.5\times$ less communication than the naïve approach of extend-then-multiply.

d) Digit Decomposition: This splits an ℓ-bit value into $c = \ell/d$ digits of d-bits. It can be realized using GC with communication $\lambda(6\ell - 2c - 2)$ bits. We propose an optimized protocol that requires communication of $\approx \lambda(c - 1)(d + 2)$ bits, that is, roughly $5\times$ lower than GC. We build on digit decomposition for an efficient protocol for MSNZB required to realize the functionality for reciprocal square root.

[4]Depth optimized GMW [49] has higher communication than GC for our functionalities.

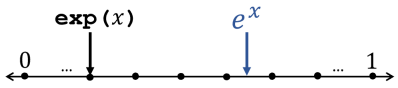

Fig. 2: The computed result exp(x) is in error of 3 ULPs from the mathematically exact result e^x. Dots denote the representable numbers.

III. PRELIMINARIES

A. Math functions and ULP errors

The math functions have irrational outputs which are impossible to represent exactly in finite number of bits. When using a finite-bit representation, like floating-point or fixed-point, the most precise implementation is the one that generates *correctly rounded* results, i.e., the output of the implementation is a representable number that is closest to the ideal $\mathbb{R}$ result. However, because of Table maker's dilemma, such implementations are computationally very expensive [45]. Consequently, standard math libraries like GNU's or Intel's `libm` don't return the correctly rounded results.

ULP error. The deviation between the finite-bit output and the exact result can be quantified in three ways: absolute error, relative error, and "units in last place" or ULPs. The former two have serious issues and the "most natural way to measure rounding error is in ulps" [48]; standard math libraries use ULPs to report the precision of their implementations [4], [111]. To see why this is the case, observe that if r is a very small real number, then the absolute error between r and $r' = 2r$, i.e., $|r - r'| = |r|$, is small as well. Hence, a low absolute error can be achieved even when every bit of the output is incorrect. Relative error, given by $|\frac{r-r'}{r}|$, remedies this situation and $r' = 2r$ leads to high relative errors irrespective of the magnitude of r. However, the relative error is undefined for $r = 0$. ULP errors have the nice property that they are always well-defined and don't grow or shrink with the magnitude of r. At a high level, the ULP error[5] between an exact real result r and the library output a is the number of representable numbers between a and r [79], [106]. We show an example in Figure 2.

Intel's SVML [4] has ULP error below 4 and MKL [111] guarantees ULP error below 1. It is important for the ULP error to be low for reusability of the library implementations as a low error gives the developers an assurance that the library is producing precise results inasmuch as the underlying representation permits.

B. Threat Model

We consider 2-party computation secure against a *static semi-honest* adversary running in probabilistic polynomial time. That is, we consider a computationally bounded adversary $\mathcal{A}$ that corrupts one of the parties at the beginning of the protocol execution, follows the protocol specification, but tries to learn additional information about the honest party's input.

[5]See [48] for the formal definition of ULPs.

We argue security using the simulation paradigm [26], [49], [81]. For any function f to be computed, consider following two interactions: a real interaction where P_0 and P_1 interact using the protocol specification in the presence of $\mathcal{A}$ and the environment $\mathcal{Z}$ and the ideal interaction where P_0, P_1 send their inputs to the trusted functionality $\mathcal{F}$ that computes f and sends the outputs to the parties. We argue that for every real adversary $\mathcal{A}$, there is an ideal adversary $\mathcal{S}$ such that no environment $\mathcal{Z}$ interacting externally with the adversaries can distinguish between real and ideal interactions. Our protocols invoke several sub-protocols and for ease of exposition we describe them using the *hybrid model*, which is the same as a real interaction except that the sub-protocol executions are replaced with calls to the corresponding trusted functionalities – protocol invoking $\mathcal{F}$ is said to be in the $\mathcal{F}$-hybrid model.

C. Notation

Let λ be computational security parameter. Uppercase L, M, N denote $2^\ell, 2^m, 2^n$, respectively. $[k]$ refers to the set $\{0, \ldots, k-1\}$. $\mathbf{1}\{b\}$ denotes the indicator function that is 1 when b is true, and 0 otherwise. We use the natural one-to-one correspondence between $\{0,1\}^\ell$ and $\mathbb{Z}_L$. Consider the lossless lifting operators ζ_ℓ that maps an element of ring $\mathbb{Z}_L$ to $\mathbb{Z}$ and $\zeta_{\ell,m}$ for $m \geqslant \ell$ that maps an element of ring $\mathbb{Z}_L$ to $\mathbb{Z}_M$. For brevity, we suppress these operations when their unambiguous use can be deduced from the context. For an element $x \in \mathbb{Z}_L$, $\mathsf{int}(x)$ and $\mathsf{uint}(x)$ refer to the signed and unsigned values in $\mathbb{Z}$ respectively, where the signed case corresponds to the 2's complement representation. $\mathsf{uint}(x)$ is defined as $\zeta_\ell(x)$ and $\mathsf{int}(x) = \mathsf{uint}(x) - \mathsf{MSB}(x) \cdot L$, where $\mathsf{MSB}(x) = \mathbf{1}\{x \geqslant 2^{\ell-1}\}$ is the most significant bit. For $x, y \in \mathbb{Z}_L$, $\mathsf{wrap}(x, y, L)$ is 1 if $x + y \geqslant L$ over $\mathbb{Z}$ and 0 otherwise. Finally, consider the operator $*_m : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_M$ where $x *_m y = x \cdot y \bmod M$. When one or both inputs are from some integer ring $\mathbb{Z}_L$, we use $\mathsf{uint}()$ and $\mathsf{int}()$ to map the element to $\mathbb{Z}$.

Fixed-Point Representation. We encode real numbers as elements in $\mathbb{Z}_L$ using their fixed-point representation. Fixed-point representation in $\mathbb{Z}_L$ defines 2 variables, ℓ and s, where ℓ is the *bitwidth*, s is the resolution (or, fractional part bitwidth) referred to as the *scale* and $\ell - s$ is the bitwidth for the integer part. A real number $x \in \mathbb{R}$ is encoded into its fixed-point representation $\hat{x} \in \mathbb{Z}_L$ with bitwidth ℓ and scale s as $\hat{x} = \mathsf{Fix}(x, \ell, s) = \lfloor x \cdot 2^s \rfloor \bmod L$. The reverse mappings from fixed-point representation to reals are $\mathsf{urt}_{(\ell,s)}(a) = \mathsf{uint}(a)/2^s$ for unsigned numbers and $\mathsf{srt}_{(\ell,s)}(a) = \mathsf{int}(a)/2^s$ for signed numbers, where division is over $\mathbb{R}$.

D. Cryptographic Primitives

Secret Sharing. We use 2-out-of-2 additive secret sharing schemes over different power-of-2 rings [16], [107]. For $x \in \mathbb{Z}_L$, we denote its shares by $\langle x \rangle^\ell = (\langle x \rangle^\ell_0, \langle x \rangle^\ell_1)$ such that $x = \langle x \rangle^\ell_0 + \langle x \rangle^\ell_1 \bmod L$ and P_b holds $\langle x \rangle^\ell_b$ for $b \in \{0,1\}$. When $\ell = 1$, i.e., over $\mathbb{Z}_2$, we use $\langle x \rangle^B$ to denote boolean shares. In our protocols, we write "P_0 & P_1 hold $\langle x \rangle^\ell$." to denote that P_b holds $\langle x \rangle^\ell_b$ for $b \in \{0,1\}$.

Oblivious Transfer. Consider 2-party functionality 1-out-of-k oblivious transfer (OT) denoted by $\binom{k}{1}$-OT_ℓ, where one party is the sender with k ℓ-bit messages $x_0, \ldots, x_{k-1} \in \{0,1\}^\ell$ and the other party is the receiver with an index $j \in [k]$. The receiver learns x_j as the output, and the sender learns nothing. We realize this functionality using the OT extension protocol from [70], which optimizes and generalizes the protocol from [62]. Additionally, we use the 1-out-of-2 correlated OT (COT) functionality $\binom{2}{1}$-COT_ℓ, which is defined as follows: sender inputs a correlation $x \in \mathbb{Z}_L$, receiver inputs a choice bit $j \in \{0,1\}$, and the functionality outputs a random element $r \in \mathbb{Z}_L$ to the sender and $-r + j \cdot x$ to the receiver. We instantiate this functionality with the COT protocol from [11]. Excluding the one-time setup cost for the base OTs, $\binom{k}{1}$-OT_ℓ and $\binom{2}{1}$-COT_ℓ require $2\lambda + k\ell$ and $\lambda + \ell$ bits of communication, respectively, and execute in 2 rounds[6]. For the special case of $k = 2$, $\binom{2}{1}$-OT_ℓ requires $\lambda + 2\ell$ bits of communication [11].

E. 2PC Functionalities

For a 2-party functionality $\mathcal{F}$, we say that "P_0 & P_1 invoke $\mathcal{F}(x, y)$ to learn $\langle z \rangle^\ell$" to mean that P_0 with input x and P_1 with input y invoke $\mathcal{F}$ and learn arithmetic shares of z over $\mathbb{Z}_L$, i.e., P_0 gets $\langle z \rangle^\ell_0$ and P_1 gets $\langle z \rangle^\ell_1$. We write "$\mathcal{F}(\langle x \rangle^\ell)$" to mean that $\mathcal{F}$ takes $\langle x \rangle^\ell_0$ from P_0 and $\langle x \rangle^\ell_1$ from P_1. In our protocols, we use the following 2-party functionalities.

- *Millionaires'/Wrap*: The ℓ-bit Millionaires' functionality, $\mathcal{F}^\ell_{\mathsf{Mill}}$ takes as input $x \in \{0,1\}^\ell$ from P_0 and $y \in \{0,1\}^\ell$ from P_1 and returns $\langle z \rangle^B$ such that $z = \mathbf{1}\{x < y\}$. The ℓ-bit wrap functionality, $\mathcal{F}^\ell_{\mathsf{Wrap}}$ on same inputs returns $\langle z \rangle^B$ such that $z = \mathsf{wrap}(x, y, L)$. Note that $\mathcal{F}^\ell_{\mathsf{Wrap}}(x, y) = \mathcal{F}^\ell_{\mathsf{Mill}}(L - 1 - x, y)$. Recently, [99] gave an efficient protocol for $\mathcal{F}^\ell_{\mathsf{Mill}}$ with communication less than[7] $\lambda\ell + 14\ell$ bits with $\log \ell$ rounds.
- AND: The functionality $\mathcal{F}_{\mathsf{AND}}$ takes as input $(\langle x \rangle^B, \langle y \rangle^B)$ and returns $\langle x \wedge y \rangle^B$. $\mathcal{F}_{\mathsf{AND}}$ can be realized using Beaver bit-triples [15] and [99] gave a protocol for $\mathcal{F}_{\mathsf{AND}}$ with $\lambda + 20$ or 148 bits[8] of total communication.
- *Boolean to Arithmetic (B2A)*: The ℓ-bit B2A functionality, $\mathcal{F}^\ell_{\mathsf{B2A}}$, takes boolean shares $\langle x \rangle^B$ and outputs arithmetic shares of the same value, i.e., $\langle x \rangle^\ell$. We use the COT based protocol from [99] with communication $\lambda + \ell$ bits.
- *Multiplexer (MUX)*: The ℓ-bit MUX functionality, $\mathcal{F}^\ell_{\mathsf{MUX}}$, takes as input $\langle x \rangle^B$ and $\langle y \rangle^\ell$ and outputs $\langle z \rangle^\ell$ such that $z = y$ if $x = 1$ and 0 otherwise. We provide an optimized protocol that reduces communication from $2(\lambda + 2\ell)$ [99] to $2(\lambda + \ell)$ (see Appendix A).
- *Lookup Table (LUT)*: The LUT functionality for table T with M entries of n-bits each, $\mathcal{F}^{T,m,n}_{\mathsf{LUT}}$ takes as

[6]Recently, MOTION [23] gave a COT protocol with similar communication and overall 2 rounds. However, their protocol requires only a single round of communication assuming precomputed ROT correlations. The total round complexity of some of our protocols can benefit from this COT.

[7]For ease of exposition, we use this rough upper bound to compute an upper bound of communication of most of our protocols.

[8]The best known communication for $\mathcal{F}_{\mathsf{AND}}$ is 138 bits [42], however, its implementation isn't available.

input $\langle x\rangle^m$ and outputs $\langle z\rangle^n$ such that $z = T[x]$. It can be realized using a single call to $\binom{M}{1}$-OT_n with communication $2\lambda + Mn$ bits [42].

IV. Building Block Protocols

In this section, we describe our building block protocols that we combine later to obtain protocols for math library functions in Section V. Our protocols extensively use the existing 2PC functionalities described in Section III-E. In addition, they invoke the functionality $\mathcal{F}^{\ell}_{\mathsf{Wrap\&All1s}}$ that takes as input $x \in \{0,1\}^\ell$ from P_0 and $y \in \{0,1\}^\ell$ from P_1 and outputs $(\langle w\rangle^B || \langle e\rangle^B)$ such that $w = \mathsf{wrap}(x, y, L)$ and $e = \mathbf{1}\{(x + y \bmod L) = L - 1\}$. We show that this functionality can be realized with nearly the same cost as $\mathcal{F}^{\ell}_{\mathsf{Wrap}}$ by making a white-box use of the protocol for $\mathcal{F}^{\ell}_{\mathsf{Mill}}$ from [99] (Appendix B). The resulting protocol has $\log \ell$ rounds and at most $\lambda\ell + 14\ell$ bits of communication. Below we describe our protocols for extension, truncation, multiplication, digit decomposition and MSB-to-wrap optimization that applies extensively to our math functionalities.

A. Zero Extension and Signed Extension

Zero and signed extension functions are used to extend the bitwidths of unsigned and signed numbers, respectively. More precisely, for an m-bit number $x \in \mathbb{Z}_M$, we define zero extension (resp. signed extension) to n-bits ($n > m$) by $y = \mathsf{ZExt}(x, m, n) \in \mathbb{Z}_N$ (resp. $y = \mathsf{SExt}(x, m, n) \in \mathbb{Z}_N$), such that $\mathsf{uint}(y) = \mathsf{uint}(x)$ (resp. $\mathsf{int}(y) = \mathsf{int}(x)$) holds. In Algorithm 1, we describe our protocol for $\mathcal{F}^{m,n}_{\mathsf{ZExt}}$ that takes as input $\langle x\rangle^m$ and outputs $\langle y\rangle^n$, where $y = \mathsf{ZExt}(x, m, n)$. This protocol requires $\log m + 2$ rounds and less than $\lambda(m+1) + 13m + n$ bits of communication.

Correctness of our protocol can be argued as follows: By correctness of $\mathcal{F}^{m}_{\mathsf{Wrap}}$ and $\mathcal{F}^{n-m}_{\mathsf{B2A}}$, it holds that $w = \mathsf{wrap}(\langle x\rangle^m_0, \langle x\rangle^m_1, M)$ and $y = \sum_{b=0}^{1}(\langle x\rangle^m_b - M \cdot \langle w\rangle^{n-m}_b) \bmod N$. Over $\mathbb{Z}$, $w = \langle w\rangle^{n-m}_0 + \langle w\rangle^{n-m}_1 - 2^{n-m} \cdot \mathsf{wrap}(\langle w\rangle^{n-m}_0, \langle w\rangle^{n-m}_1, 2^{n-m})$. Thus, $M *_n w = M *_n (\langle w\rangle^{n-m}_0 + \langle w\rangle^{n-m}_1)$. Also, over $\mathbb{Z}$, $x = \langle x\rangle^m_0 + \langle x\rangle^m_1 - w \cdot M$. Hence, $x \bmod N = y$.

Our protocol for signed extension, i.e., $\mathcal{F}^{m,n}_{\mathsf{SExt}}$, uses the following equation over $\mathbb{Z}$:

$$\mathsf{int}(x) = x' - 2^{m-1}, \text{ for } x' = x + 2^{m-1} \bmod M. \quad (1)$$

This gives[9] $\mathsf{SExt}(x, m, n) = \mathsf{ZExt}(x', m, n) - 2^{m-1}$.

As a baseline, one can use garbled circuits (GC) to realize zero and signed-extensions with communication cost of $\lambda(4m+2n-4)$ bits, i.e., roughly $6\times$ the cost of our protocols.

B. Truncation

We consider four types of truncation operations for ring $\mathbb{Z}_L$ as follows: We denote the logical and arithmetic right-shift operators by $\gg_L$ and $\gg_A$, respectively, whose inputs are outputs are in $\mathbb{Z}_L$. Next, we define $\mathsf{TR}(x, s)$ (Truncate & Reduce x by s-bits) that takes inputs in $\mathbb{Z}_L$, drops the lower s-bits from the bit-representation of x and outputs the truncated value in smaller ring, $\mathbb{Z}_{2^{\ell-s}}$. Additionally, our benchmarks also require the C-style division (quotients are rounded towards 0) where the divisor is a power-of-2.

[9] A similar relation was used in [44] for truncation.

Algorithm 1 Zero Extension, $\Pi^{m,n}_{\mathsf{ZExt}}$:

Input: P_0 & P_1 hold $\langle x\rangle^m$.
Output: P_0 & P_1 get $\langle y\rangle^n$ for $y = \mathsf{ZExt}(x, m, n)$.

1: P_0 & P_1 invoke $\mathcal{F}^{m}_{\mathsf{Wrap}}(\langle x\rangle^m_0, \langle x\rangle^m_1)$ and learn $\langle w\rangle^B$.
2: P_0 & P_1 invoke $\mathcal{F}^{n-m}_{\mathsf{B2A}}(\langle w\rangle^B)$ and learn $\langle w\rangle^{n-m}$.
3: For $b \in \{0,1\}$, P_b outputs $\langle y\rangle^n_b = \langle x\rangle^m_b - M *_n \langle w\rangle^{n-m}_b$.

Algorithm 2 Logical Right Shift, $\Pi^{\ell,s}_{\mathsf{LRS}}$:

Input: P_0 & P_1 hold $\langle x\rangle^\ell$.
Output: P_0 & P_1 get $\langle x \gg_L s\rangle^\ell$.

1: For $b \in \{0,1\}$, P_b parses $\langle x\rangle^\ell_b$ as an ℓ-bit string $u_b||v_b$, where $u_b \in \{0,1\}^{\ell-s}$ and $v_b \in \{0,1\}^s$.
2: P_0 & P_1 invoke $\mathcal{F}^{s}_{\mathsf{Wrap}}(v_0, v_1)$ and learn $\langle c\rangle^B$.
3: P_0 & P_1 invoke $\mathcal{F}^{\ell-s}_{\mathsf{Wrap\&All1s}}(u_0, u_1)$ and learn $\langle d\rangle^B || \langle e\rangle^B$.
4: P_0 & P_1 invoke $\mathcal{F}_{\mathsf{AND}}(\langle c\rangle^B, \langle e\rangle^B)$ and learn $\langle t\rangle^B$.
5: For $b \in \{0,1\}$, P_b sets $\langle w\rangle^B_b = \langle d\rangle^B_b \oplus \langle t\rangle^B_b$.
6: P_0 & P_1 invoke $\mathcal{F}^{\ell}_{\mathsf{B2A}}(\langle c\rangle^B)$ and learn $\langle c\rangle^\ell$.
7: P_0 & P_1 invoke $\mathcal{F}^{s}_{\mathsf{B2A}}(\langle w\rangle^B)$ and learn $\langle w\rangle^s$.
8: For $b \in \{0,1\}$, P_b outputs $u_b - 2^{\ell-s} *_\ell \langle w\rangle^s_b + \langle c\rangle^\ell_b$.

Logical Right Shift. In Algorithm 2, we describe our protocol for $\mathcal{F}^{\ell,s}_{\mathsf{LRS}}$ that takes as input $\langle x\rangle^\ell$ and outputs $\langle x \gg_L s\rangle^\ell$. The idea is as follows: Consider $x \in \mathbb{Z}_L$ and $\langle x\rangle^\ell$. Also, for $b \in \{0,1\}$, let $\langle x\rangle^\ell_b = u_b||v_b$ where $u_b \in \{0,1\}^{\ell-s}$ and $v_b \in \{0,1\}^s$. Then, it can be shown that $x \gg_L s = u_0 + u_1 - 2^{\ell-s} \cdot \mathsf{wrap}(\langle x\rangle^\ell_0, \langle x\rangle^\ell_1, L) + \mathsf{wrap}(v_0, v_1, 2^s)$ [21]. A simple protocol for $\mathcal{F}^{\ell,s}_{\mathsf{LRS}}$ computes shares of wrap terms over ℓ-bits and s-bits separately. We further optimize this protocol using the following lemma (proof appears in Appendix C):

Lemma 1. *Let $x \in \mathbb{Z}_L$, $\langle x\rangle^\ell$ be shares of x and for $b \in \{0,1\}$, $\langle x\rangle^\ell_b = u_b||v_b$, where $u_b \in \{0,1\}^{\ell-s}$ and $v_b \in \{0,1\}^s$. Define $c = \mathsf{wrap}(v_0, v_1, 2^s)$, $d = \mathsf{wrap}(u_0, u_1, 2^{\ell-s})$, $e = \mathbf{1}\{u_0 + u_1 \bmod 2^{\ell-s} = 2^{\ell-s} - 1\}$ and $w = \mathsf{wrap}(\langle x\rangle^\ell_0, \langle x\rangle^\ell_1, L)$, then it holds that $w = d \oplus (c \wedge e)$.*

Using this lemma, our protocol only uses wrap computations over $\ell - s$ and s bits and a call to $\mathcal{F}_{\mathsf{AND}}$ functionality. As another optimization, while invoking $\mathcal{F}_{\mathsf{B2A}}$ on shares of w, we go to arithmetic shares over $\mathbb{Z}_{2^s}$ (and not $\mathbb{Z}_L$). Overall communication cost is less than $\lambda(\ell+3) + 15\ell + s + 20$ and rounds required are $\log \ell + 3$.

Arithmetic Right Shift. Our protocol for $\mathcal{F}^{\ell,s}_{\mathsf{ARS}}$ that outputs $\langle x \gg_A s\rangle^\ell$ builds upon $\mathcal{F}^{\ell,s}_{\mathsf{LRS}}$ using the relation [44]: $x \gg_A s = x' \gg_L s - 2^{\ell-s-1}$, where $x' = x + 2^{\ell-1}$. Hence, it has the same cost as $\Pi^{\ell,s}_{\mathsf{LRS}}$. Prior state-of-the-art protocol for arithmetic right shift is from CrypTFlow2 [99] that runs in $\log \ell + 2$ rounds with communication $\lambda(\ell + s + 2) + 19\ell + 14s$ bits. Note that unlike our protocol, its communication grows

multiplicatively in λ with both ℓ and s.

Truncate and Reduce. Many of our protocols can benefit from truncate and reduce to the smaller ring over logical/arithmetic right shift operations that output shares in the original ring. At a high level, our protocol for $\mathcal{F}^{\ell,s}_{\mathsf{TR}}$ that outputs $\langle \mathsf{TR}(x,s)\rangle^{\ell-s}$ is as follows: Using the above notation, $\mathsf{TR}(x,s) = u_0 + u_1 + \mathsf{wrap}(v_0, v_1, 2^s)$. Hence, we can skip the computation of shares of w, i.e., steps 3–7 can be skipped. Overall communication is $\lambda(s+1) + \ell + 13s$ bits. The best solution using prior techniques is: $\mathsf{TR}(x,s) = (x \gg_A s) \bmod 2^{\ell-s}$, which would incur the same cost as the state-of-the-art ARS protocol [99], i.e., $\lambda(\ell+s+2) + 19\ell + 14s$ bits.

Division by power-of-2. In addition to arithmetic right shift, the fixed-point code for ML benchmarks require C-style division by power-of-2 to preserve model accuracy. Consider the functionality $\mathcal{F}^{\ell,s}_{\mathsf{DivPow2}}$ that takes $\langle x\rangle^\ell$ as input and outputs $\langle z\rangle^\ell$ such that $z = \lceil \mathsf{int}(x)/2^s \rceil \bmod L$ for $z < 0$ and $z = \lfloor \mathsf{int}(x)/2^s \rfloor \bmod L$ for $z \geqslant 0$. We give an overview of our protocol in Appendix C that requires roughly $\lambda(\ell+2s+4)$ bits of communication. To the best of our knowledge, no prior work explicitly builds a protocol for this functionality. A garbled circuits implementation, costs $\lambda(8\ell+2s-6)$ bits.

C. Multiplication with non-uniform bitwidths

Our machine learning models as well as math library functions (see Section V) use multiplication operation with operands of different bit-widths that outputs a value in the larger ring. Below, we describe these functions and their protocols for both the unsigned and the signed case.

Unsigned Multiplication with non-uniform bitwidths. Consider the functionality $\mathcal{F}^{m,n}_{\mathsf{UMult}}$ that takes $\langle x\rangle^m$ and $\langle y\rangle^n$ as input and returns $\langle z\rangle^\ell$, where $z = x *_\ell y$, for $\ell = m+n$. In contrast, all prior works on secure inference [64], [83], [90], [92], [99], [102], use $m = n = \ell$. A naïve way to realize this functionality is to first extend both the inputs to ℓ-bits and then use standard multiplication, i.e., multiply $\mathcal{F}^{m,\ell}_{\mathsf{ZExt}}(\langle x\rangle^m)$ and $\mathcal{F}^{n,\ell}_{\mathsf{ZExt}}(\langle y\rangle^n)$ using existing protocols for uniform bit-widths. We give a new custom protocol for multiplying values of non-uniform bitwidths that beats this naïve approach by roughly $1.5\times$. Our protocol builds on the functionality $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}} : \mathbb{Z}_M \times \mathbb{Z}_N \to \mathbb{Z}_L \times \mathbb{Z}_L$ that is defined as $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}(x,y) = \langle z\rangle^\ell$, where $z = x *_\ell y$. We describe our protocol for $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$ in Appendix D1 that carefully uses $\binom{2}{1}$-OT (to minimize overall communication) similar to the techniques of generating Beaver triples [15]. The communication complexity of this protocol is $\mu(\lambda + \mu/2 + 1/2) + mn$, where $\mu = \mathsf{min}(m,n)$.

By definition of $*_\ell$, we wish to compute $\mathsf{uint}(x) \cdot \mathsf{uint}(y) \bmod L$, where $\ell = m+n$. Let $x = x_0 + x_1 \bmod M$ and $y = y_0 + y_1 \bmod N$. Algorithm 3 gives our protocol for $\mathcal{F}^{m,n}_{\mathsf{UMult}}$ that builds on the following: Over $\mathbb{Z}$,

$$\begin{aligned}\mathsf{uint}(x)\cdot\mathsf{uint}(y) &= (x_0 + x_1 - 2^m w_x)\cdot(y_0 + y_1 - 2^n w_y)\\ &= x_0y_0 + x_1y_1 + x_0y_1 + x_1y_0\\ &\quad - 2^m w_x y - 2^n w_y x - 2^\ell w_x w_y, \qquad (2)\end{aligned}$$

where $w_x = \mathsf{wrap}(x_0, x_1, M)$ and $w_y = \mathsf{wrap}(y_0, y_1, N)$. Taking a $\bmod L$, removes the last term. In the protocol, party P_b computes $x_b y_b$ as $(x_b *_\ell y_b)$ locally and invokes $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$ to compute shares of cross-terms $x_b y_{1-b}$. Wraps are computed using $\mathcal{F}_{\mathsf{Wrap}}$ and multiplied to values using $\mathcal{F}_{\mathsf{MUX}}$.

The communication complexity of our protocol is roughly $\lambda(3\mu+\nu) + \mu(\mu+2\nu) + 16(m+n)$ where $\mu = \mathsf{min}(m,n)$ and $\nu = \mathsf{max}(m,n)$. In contrast, communication complexity of naïve approach of extend-then-multiply that uses our optimized protocols for extension is roughly $3\lambda(\mu+\nu)+(m+n)^2+15(m+n)$, i.e., roughly $1.5\times$ more than our new protocol.

We note that the same ideas also work for the setting $\ell < m+n$ by using an appropriate protocol for $\mathcal{F}^{m,n,\ell}_{\mathsf{CrossTerm}}$ with specific value of ℓ. Similarly, we define the multiplication functionality $\mathcal{F}^{m,n,\ell}_{\mathsf{UMult}}$ which internally invokes $\mathcal{F}^{m,n,\ell}_{\mathsf{CrossTerm}}$, where the additional superscript denotes the bitwidth of the output. Our protocols for math library functions also uses this setting for better efficiency.

Signed Multiplication with non-uniform bitwidths. Consider the functionality $\mathcal{F}^{m,n}_{\mathsf{SMult}}$ that takes $\langle x\rangle^m$ and $\langle y\rangle^n$ as input and returns $\langle z\rangle^\ell$, where $z = \mathsf{int}(x) *_\ell \mathsf{int}(y)$, for $\ell = m+n$. Let $x' = x + 2^{m-1} \bmod M, y' = y + 2^{n-1} \bmod N$ such that $x' = x'_0 + x'_1 \bmod M$ and $y' = y'_0 + y'_1 \bmod N$. Our protocol for $\mathcal{F}^{m,n}_{\mathsf{SMult}}$ builds on the following equations over $\mathbb{Z}$:

$$\begin{aligned}\mathsf{int}(x)\cdot\mathsf{int}(y) &= (x' - 2^{m-1})\cdot(y' - 2^{n-1}) \quad \text{from Eq. 1}\\ &= x'\cdot y' - 2^{m-1}y' - 2^{n-1}x' + 2^{m+n-2}\\ &= x'\cdot y' - 2^{m-1}(y'_0 + y'_1 - 2^n w_{y'})\\ &\quad - 2^{n-1}(x'_0 + x'_1 - 2^m w_{x'}) + 2^{m+n-2},\end{aligned}$$

where $w_{x'} = \mathsf{wrap}(x'_0, x'_1, M), w_{y'} = \mathsf{wrap}(y'_0, y'_1, N)$.

In the protocol, parties can compute the shares of x', y' locally. All terms in the final expression can be computed and added locally except $z_1 = x'y'$ and $z_2 = 2^{\ell-1}(w_{x'} + w_{y'})$. Since the final expression needs to be computed $\bmod L$, we can compute shares of z_1 in $\mathbb{Z}_L$ using a call to $\Pi^{m,n}_{\mathsf{UMult}}$. We piggyback the computation of boolean shares of $w_{x'}$ and $w_{y'}$ on $\Pi^{m,n}_{\mathsf{UMult}}$, which already computes them in steps 5&6. Note

Algorithm 3 Unsigned Multiplication, $\Pi^{m,n}_{\mathsf{UMult}}$:

Input: P_0 & P_1 hold $\langle x\rangle^m$ and $\langle y\rangle^n$.
Output: P_0 & P_1 get $\langle z\rangle^\ell$, where $z = x *_\ell y$ and $\ell = m+n$.

1: For $b \in \{0,1\}$, let $x_b = \langle x\rangle^m_b$ and $y_b = \langle y\rangle^n_b$.
2: P_0 and P_1 invoke the following functionalities.
3: $\quad\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}(x_0, y_1)$ and learn $\langle c\rangle^\ell$.
4: $\quad\mathcal{F}^{n,m}_{\mathsf{CrossTerm}}(y_0, x_1)$ and learn $\langle d\rangle^\ell$.
5: $\quad\mathcal{F}^{m}_{\mathsf{Wrap}}(x_0, x_1)$ to learn $\langle w_x\rangle^B$.
6: $\quad\mathcal{F}^{n}_{\mathsf{Wrap}}(y_0, y_1)$ to learn $\langle w_y\rangle^B$.
7: $\quad\mathcal{F}^{m}_{\mathsf{MUX}}(\langle w_y\rangle^B, \langle x\rangle^m)$ to learn $\langle g\rangle^m$.
8: $\quad\mathcal{F}^{n}_{\mathsf{MUX}}(\langle w_x\rangle^B, \langle y\rangle^n)$ to learn $\langle h\rangle^n$.
9: P_b outputs $x_b *_\ell y_b + \langle c\rangle^\ell_b + \langle d\rangle^\ell_b - N *_\ell \langle g\rangle^m_b - M *_\ell \langle h\rangle^n_b$ for $b \in \{0,1\}$.

that $2^{\ell-1}w_{x'} = 2^{\ell-1}(\langle w_{x'}\rangle_0^B + \langle w_{x'}\rangle_1^B - 2\langle w_{x'}\rangle_0^B\langle w_{x'}\rangle_1^B)$ and taking a mod L gets rid of the last term. Hence, $2^{\ell-1}(\langle w_{x'}\rangle^B + \langle w_{y'}\rangle^B)$ are correct arithmetic shares of z_2 in $\mathbb{Z}_L$. Thus, we can do signed multiplication with a single call to $\Pi_{\mathsf{UMult}}^{m,n}$ and no additional cost.

We also consider the signed-multiplication functionality $\mathcal{F}_{\mathsf{SMult}}^{m,n,\ell}$, where the output bitwidth $\ell < m+n$. The above discussion on signed-multiplication holds in this case as well, and thus, $\Pi_{\mathsf{SMult}}^{m,n,\ell}$ has the same cost as $\Pi_{\mathsf{UMult}}^{m,n,\ell}$.

Matrix Multiplication and Convolutions. Two commonly used operations in machine learning are matrix multiplications and convolutions that build on element-wise multiplications. Consider matrix multiplication of $A \in \mathbb{Z}_M^{d_1\times d_2}$ and $B \in \mathbb{Z}_N^{d_2\times d_3}$, where we would like to use our protocol for $\mathcal{F}_{\mathsf{UMult}}^{m,n}$. Now, each element in the output product matrix is a result of d_2 multiplications and d_2-1 additions and even when the result of multiplication is stored in the larger ring $\mathbb{Z}_L$, $\ell = m+n$, the value can overflow due to additions. One way to avoid this overflow is to extend the result of element-wise products by $e = \lceil \log d_2 \rceil$ bits and then do the additions. However, this method is quite expensive as the number of extensions needed would be $d_1d_2d_3$. We significantly reduce this cost as follows: Since the cost of $\mathcal{F}_{\mathsf{CrossTerm}}$ depends on the smaller of the two bitwidths, we extend the values in the matrix of larger bitwidth by e bits. Then we perform the matrix multiplications into $\mathbb{Z}_{2^{m+n+e}}$, ensuring that there are no overflows. Moreover, similar to the OT-based matrix multiplication from prior works [92], [99], we also exploit the multi-use of input matrix elements to optimize the cost of computing (matrix) cross-terms in our protocol. Our protocol has communication complexity roughly $\lambda\left(3d_1d_2\left(m+2\right)+d_2d_3\left(n+2\right)\right) + d_1d_2d_3\left((2m+4)(n+e)+m^2+5m\right)$ bits for $m \leqslant n$ ignoring lower order terms. We describe our protocol formally in Appendix D2 along with exact communication complexity. Above ideas easily extend to computing convolutions as well.

Multiply and Truncate. In most of our protocols, we first invoke $\mathcal{F}_{\mathsf{SMult}}^{m,n,\ell}$ followed by $\mathcal{F}_{\mathsf{TR}}^{\ell,s}$, where $\ell \leqslant m+n$. Hence, for ease of exposition, we define the functionality $\mathcal{F}_{\mathsf{SMultTR}}^{m,n,\ell,s}$ for signed multiplication and truncate-reduce that takes $\langle x\rangle^m$ and $\langle y\rangle^n$ as input and returns $\langle z'\rangle^{\ell-s}$ such that $z = \mathsf{int}(x) *_\ell \mathsf{int}(y)$ and $z' = \mathsf{TR}(z,s)$.

D. Digit Decomposition and MSNZB

We consider the functionality $\mathcal{F}_{\mathsf{DigDec}}^{\ell,\{d_i\}_{i\in[c]}}$ that decomposes an ℓ-bit number into c sub-strings or digits of lengths $\{d_i\}$. More formally, $\mathcal{F}_{\mathsf{DigDec}}^{\ell,\{d_i\}_{i\in[c]}}$ takes $\langle x\rangle^\ell$ as input and outputs $\langle z_{c-1}\rangle^{d_{c-1}}, \ldots, \langle z_0\rangle^{d_0}$ such that $x = z_{c-1}||\ldots||z_0$.

For an ℓ-bit integer x, $\mathsf{MSNZB}(x)$ refers to the index of the most significant non-zero-bit. That is, $\mathsf{MSNZB}(x) = k \in [\ell]$, if $x_k = 1$ and $x_j = 0$ for all $j > k$. Consider the functionality $\mathcal{F}_{\mathsf{MSNZB}}^{\ell}$ that takes as input $\langle x\rangle^\ell$ and outputs $\{\langle z_i\rangle^B\}_{i\in[\ell]}$ such that $z_i = 1$ if $\mathsf{MSNZB}(x) = i$ and 0 otherwise.

We describe the protocols for $\mathcal{F}_{\mathsf{DigDec}}^{\ell,\{d_i\}_{i\in[c]}}$ and $\mathcal{F}_{\mathsf{MSNZB}}^{\ell}$ in Appendix E and F, respectively.

E. MSB-to-Wrap Optimization

Our protocols above for extension, truncation and multiplication make use of the following step: Parties P_0, P_1 hold $\langle x\rangle^\ell$ and compute $\langle w\rangle^B$, where $w = \mathsf{wrap}(\langle x\rangle_0^\ell, \langle x\rangle_1^\ell, L)$. This is either computed through an explicit call to $\mathcal{F}_{\mathsf{Wrap}}^{\ell}$ (e.g., extension and multiplication) or computed via wrap of lower and upper bits (e.g., truncation). We show that shares of w can be computed with much less communication and rounds if the parties either know the $m_x = \mathsf{MSB}(x)$ in the clear or shared form. The MSB refers to the most significant bit of a number. In our math library implementations in Section V, this condition is true for almost all invocations. For instance, in exponential, when multiplying the values from multiple LUTs, we know that all operands are positive, i.e., MSB of all inputs to multiplication is 0. We call this optimization *MSB-to-Wrap* and the idea is as follows: We can write $w = ((1 \oplus m_x) \wedge (m_0 \oplus m_1)) \oplus (m_0 \wedge m_1)$, where $m_b = \mathsf{MSB}(\langle x\rangle_b^\ell)$ for $b \in \{0,1\}$. With this, given shares of m_x, boolean shares of w can be computed using a single call to $\binom{4}{1}$-OT_1, i.e., $2\lambda+4$ bits of communication and 2 rounds. Also, when m_x is publicly known, this can be computed using $\binom{2}{1}$-OT_1, i.e., $\lambda+2$ bits. The cost of our protocols with above optimization are provided in Table V.

V. Math Library Functions

In this section, we provide our cleartext implementations for math functions exponential, sigmoid, tan hyperbolic (tanh), and reciprocal square root as well as the protocols for the same. Note that these functions are impossible to implement exactly using finite-bit arithmetic, and hence, our implementations realize them approximately (Section V-D). Below, we use the notation from Section III-C and Section III-D. For a mathematical function f, we consider the functionality $\mathcal{F}_f^{m,s,n,s'}$ that takes as input the shares $\langle x\rangle^m$ and outputs $\langle y\rangle^n$ such that $\mathsf{srt}_{(n,s')}(y) \approx f(\mathsf{srt}_{(m,s)}(x))$.

Our math function implementations rely on functions discussed in Section IV, and we recall some of them here. We denote signed-extension of an m-value to an n-value by $\mathsf{SExt}(x,m,n)$ with $n > m$. Next, we denote truncate-and-reduce by s-bits using $\mathsf{TR}(x,s)$ that takes a value x of, say, ℓ-bits, drops lower s bits and returns the corresponding $(\ell-s)$-bit value. Finally, we use a signed multiplication where the operands and the output can have unequal bitwidths. It is denoted by $x *_\ell y$, where x and y are, say, m and n-bit integers, respectively, and the output of multiplication is $z = \mathsf{int}(x)\cdot\mathsf{int}(y) \bmod L$.

A. Exponential

Consider the math functionality $\mathcal{F}_{\mathsf{rExp}}^{m,s,n,s'}$ with $\mathsf{rExp}(z) = e^{-z}$, $z \in \mathbb{R}^+$ described in Figure 3. Intuitively, the correctness of this functionality, i.e., $\mathsf{srt}_{(n,s')}(y) \approx \mathsf{rExp}(\mathsf{srt}_{(m,s)}(x))$, relies on $\mathsf{rExp}(\mathsf{srt}_{(m,s)}(x)) = \mathsf{rExp}(2^{d(k-1)-s}x_{k-1}) \cdot \ldots \cdot \mathsf{rExp}(2^{-s}x_0)$. Each rExp call on the RHS can be computed

approximately using a lookup table L of size 2^d with $s'+2$ bit entries of scale s'. Since the entries of the LUTs are between 0 and 1 with scale s', it is sufficient to have a bitwidth of $s'+2$. For instance, when $m=n=16$, $d=8$, and $s'=14$ we use two LUTs where first maps the upper 8 bits of x and second maps the lower 8 bits of x. Final output is computed by multiplying the two 16-bit looked up values from the two LUTs into a 32-bit number followed by an appropriate truncate and reduce operation to get 16-bit y with scale 14. We formally verify that for m, s, n, s' used in our evaluation, our choice of d ensures precise results in Section V-D.

The protocol for this functionality can be built easily relying on the protocols described in Section IV. Step 1 can be implemented by a call to the digit decomposition functionality, $\mathcal{F}_{\mathsf{DigDec}}$. The LUTs in Step 2 can be looked up using $\mathcal{F}_{\mathsf{LUT}}$ (Section III-E). These $s'+2$-bit values are multiplied using a tree-based multiplication using $\mathcal{F}_{\mathsf{SMultTR}}^{s'+2,s'+2,2s'+2,s'}$ to get an $s'+2$-bit number with scale s' in Step 3. Finally, Step 4 extends g to an n-bit value using $\mathcal{F}_{\mathsf{SExt}}^{s'+2,n}$. Table II gives our concrete numbers and compares with prior work.

Functionality $\mathcal{F}_{\mathsf{rExp}}^{m,s,n,s'}(\langle x\rangle^m)$

1) Let $x = x_{k-1}||\ldots||x_0$, $x_i \in \{0,1\}^d$, $i \in [k]$, $dk = m$.
2) For $i \in [k]$, let $L_i : \{0,1\}^d \to \mathbb{Z}_{2^{s'+2}}$ s.t. $L_i(j) = \mathsf{Fix}\left(\mathsf{rExp}(2^{di-s}j), s'+2, s'\right)$.
3) Compute $g = L_{k-1}[x_{k-1}] * \ldots * L_0[x_0]$, g has bitwidth $s'+2$ and scale s'.
4) Return $\langle y\rangle^n$ for $y = \mathsf{SExt}(g, s'+2, n)$.

Fig. 3: *The functionality* $\mathcal{F}_{\mathsf{rExp}}^{m,s,n,s'}$ *for a parameter* d.

B. Sigmoid and Tanh

Consider the math functionality $\mathcal{F}_{\mathsf{sigmoid}}^{m,s,n,s'}$ where $\mathsf{sigmoid}(z) = \frac{1}{1+e^{-z}}$ can be written as

$$\mathsf{sigmoid}(z) = \begin{cases} 0.5, & \text{if } z = 0 \\ \frac{1}{1+\mathsf{rExp}(z)}, & \text{if } z > 0 \\ \mathsf{rExp}(-z)\frac{1}{1+\mathsf{rExp}(-z)}, & \text{if } z < 0 \end{cases}$$

Hence, sigmoid can be built by extending the math functionality $\mathcal{F}_h^{m,s,n,s'}$ such that $h(z) = \frac{1}{1+\mathsf{rExp}(z)}$, $z \in \mathbb{R}^+$ described in Figure 4. This functionality calls $\mathcal{F}_{\mathsf{rExp}}$ that we described above, followed by a call to a functionality to approximate the reciprocal that we describe next.

Functionality $\mathcal{F}_h^{m,s,n,s'}(\langle x\rangle^m)$

1) $\langle u\rangle^{s'+2} \leftarrow \mathcal{F}_{\mathsf{rExp}}^{m,s,s'+2,s'}(\langle x\rangle^m)$.
2) $\langle w\rangle^{s'+2} \leftarrow \mathcal{F}_{\mathsf{Rec}}^{s'+2,s'}(\langle 2^{s'}+u\rangle^{s'+2})$.
3) Return $\mathsf{SExt}(w, s'+2, n)$.

Fig. 4: *The functionality* $\mathcal{F}_h^{m,s,n,s'}$.

For computing the reciprocal, we rely on the Goldschmidt's algorithm [50] that iterates on an initial approximation [63]. This initial approximation requires that we only compute reciprocal of values v such that $1 \leqslant \mathsf{srt}_{(\ell,s)}(v) < 2$ which is true for the case of h and sigmoid.

We describe the math functionality $\mathcal{F}_{\mathsf{Rec}}^{\ell,s}$ in Figure 5 that maps inputs v with bitwidth ℓ and scale s to outputs of same bitwidth and scale. Since $1 \leqslant \mathsf{srt}_{(\ell,s)}(v) < 2$, in Step 1, $d=1$. We use the g most significant bits of the fractional part to index into the LUT L_{rec} in Step 2 whose entries are described in [63]. The initial approximation w has bitwidth $s+1$ and scale s. If the number of Goldschmidt iterations t is set to 0, then $\mathcal{F}_{\mathsf{Rec}}$ outputs initial approximation sign extended to output bitwidth, i.e., $\mathsf{SExt}(w, s+1, \ell)$. We formally verify that for m, s, n, s' used in our evaluation, our choice of parameters for $\mathcal{F}_{\mathsf{rExp}}$ and $\mathcal{F}_{\mathsf{Rec}}$ ensures precise results for $\mathcal{F}_{\mathsf{sigmoid}}$ in Section V-D.

Note that this functionality crucially utilizes arithmetic over variable bitwidth and extension/truncation operations and these steps require our efficient protocols from Section IV. Table I gives our concrete numbers and compares with prior work.

Functionality $\mathcal{F}_{\mathsf{Rec}}^{\ell,s}(\langle v\rangle^\ell)$

Computes the initial approximation w as follows [63]:

1) $v = d||e||f$, $d \in \{0,1\}^{\ell-s}$, $e \in \{0,1\}^g$, $f \in \{0,1\}^{s-g}$.
2) $c_0||c_1 = L_{\mathsf{rec}}(e)$, $c_0 \in \{0,1\}^{g+4}$ and $c_1 \in \{0,1\}^{2g+3}$.
3) $c_2 = \mathsf{SExt}((c_0 *_{s+4} f), s+4, s+g+4)$.
4) $w' = 2^{s-g+1} *_{s+g+4} c_1 - c_2$, $w = \mathsf{TR}(w', g+3)$.

Goldschmidt's method for t iterations.

1) $p_1 = 2^s - \mathsf{TR}(v *_{2s+2} w, s)$.
2) $q_1 = 2^s + p_1$, $a_1 = q_1$.
3) For $i \in \{2, \ldots, t\}$ do
   a) $a_i = \mathsf{TR}(a_{i-1} *_{2s+2} q_{i-1}, s)$.
   b) $p_i = \mathsf{TR}(p_{i-1} *_{2s+2} p_{i-1}, s)$.
   c) $q_i = 2^s + p_i$.
4) Return $\mathsf{SExt}(a_t, s+2, \ell)$.

Fig. 5: *The functionality* $\mathcal{F}_{\mathsf{Rec}}^{\ell,s}$ *for a parameters* g, t.

Tanh. The math functionality $\mathcal{F}_{\mathsf{Tanh}}^{m,s,n,s'}$ where $\mathsf{Tanh}(z) = \frac{e^z-e^{-z}}{e^z+e^{-z}} = 2 \cdot \mathsf{sigmoid}(2z) - 1$ can be realized using $\mathcal{F}_{\mathsf{sigmoid}}$.

C. Reciprocal of Square Root

In ML, reciprocal square root is typically used to scale down vectors $\vec{u}$ of large magnitude to unit vectors by dividing each entry of the vector with $\frac{1}{\sqrt{u^T u}}$. The reciprocal square root function maps x to $\frac{1}{\sqrt{x}}$, for $x > 0$. If x is small then to avoid divide-by-zero errors a small public constant ϵ is added to x and $\frac{1}{\sqrt{x+\epsilon}}$ is computed instead. Hence, we present our mathematical functionality $\mathcal{F}_{\mathsf{rsqrt}}^{\ell,s,\ell,s'}$ in Figure 6 for the math function $\mathsf{rsqrt}(z) = \frac{1}{\sqrt{z}}$ where $z \geqslant \epsilon$.

This functionality follows a similar template of first computing an initial approximation for reciprocal square root followed by Goldschmidt's iterations. The initial approximation [10] requires $1 \leqslant x < 2$, and hence, first we perform a range reduction to map arbitrary x of the form $y.z$ to x' of the form

[10] Although we would have liked to use the initial approximation provided by [63], there seems to be some typographical errors in the published equations and we are unable to correct them.

$1.z'$ that satisfies this constraint. This requires computing the most significant non-zero bit (MSNZB) of x (Step 1). Note that $\mathsf{MSNZB}(x) = k \in [\ell]$ if $x_k = 1$ and all $x_i = 0$ for all $i > k$. The normalized value x' has bitwidth ℓ and scale $\ell - 2$. Next, we use g most significant bits of z', i.e., e and the parity of $k - s$, i.e., B, to compute the initial approximation via a lookup table L_{rsqrt} whose entries are as follows:

$$L_{\mathsf{rsqrt}}(e||B) = \mathsf{Fix}\left(\frac{1}{\sqrt{(B+1)(1+\mathsf{urt}_{(g,g)}(e))}}, g+4, g+2\right)$$

Functionality $\mathcal{F}_{\mathsf{rsqrt}}^{\ell,s,\ell,s'}(\langle x\rangle^{\ell})$

Normalizes x to x' as follows:

1) $k = \mathsf{MSNZB}(x) \in [\ell]$.
2) $A = 2^{\ell-2-k}$, $B = (s-k) \bmod 2$.
3) $C = 2^{\lceil \frac{s-k}{2} \rceil + \lfloor \frac{\ell-s-1}{2} \rfloor}$.
4) $x' = x *_{\ell} A$.

Computes the initial approximation w as follows:

1) $x' = d||e||f, d \in \{0,1\}^2, e \in \{0,1\}^g, f \in \{0,1\}^{\ell-2-g}$.
2) $w = L_{\mathsf{rsqrt}}(e||B)$, $w \in \{0,1\}^{g+4}$.

Goldschmidt's method for t iterations:

1) $x'' = \mathsf{TR}(x', \ell - 3 - s')$, $q_0 = B \ ? \ x'' : \mathsf{TR}(x', 1)$.
2) $a_0 = 2^{s'-g-2} *_{s'+2} w$, $p_0 = a_0$.
3) For $i \in \{1, \ldots, t\}$ do
   a) $Y_i = \mathsf{TR}(p_{i-1} *_{2s'+2} p_{i-1}, s')$.
   b) $q_i = \mathsf{TR}(q_{i-1} *_{2s'+2} Y_i, s')$.
   c) $p_i = 3 \cdot 2^{s'-1} - (q_i \gg_A 1)$.
   d) $a_i = \mathsf{TR}(a_{i-1} *_{2s'+2} p_i, s')$.

Uses reciprocal square root of x' to compute the same for x:

1) Return $\mathsf{TR}(a_t *_{\ell/2+s'+3} C, \lfloor \frac{\ell-s-1}{2} \rfloor) \bmod L$.

Fig. 6: *The functionality* $\mathcal{F}_{\mathsf{rsqrt}}^{\ell,s,s'}$ *for parameters* g, t.

We formally verify that for ℓ, s, s' in our evaluation, our choice of g, t ensures precise results for $\mathcal{F}_{\mathsf{rsqrt}}$ (Section V-D).

We build a protocol for $\mathcal{F}_{\mathsf{rsqrt}}$ as follows: We consider the functionality $\mathcal{F}_{\mathsf{MSNZB}}$ that outputs the shares of one-hot encoding of $\mathsf{MSNZB}(x)$ and give a protocol for the same in Appendix F. It is easy to compute the terms A, B, C using dot-products of this one-hot vector with publicly known vectors. For our initial approximation, we rely on protocols for $\mathcal{F}_{\mathsf{DigDec}}$ and $\mathcal{F}_{\mathsf{LUT}}$. The Goldschmidt's iterations crucially utilize arithmetic over variable bitwidth and truncation operations and each of these steps require our efficient protocols from Section IV. Table II gives our concrete numbers and compares with prior work.

D. Formal verification of our Math functionalities

It is desirable for math libraries to have a formal proof of correctness about their purported numerical precision. Such a proof establishes that for all possible inputs, the ULP error (Section III) between the math implementation and the exact real result is small. For small bitwidths (e.g. $\leqslant 32$) that are used in ML (Section VI-C), it is tractable to prove these bounds on ULP error using *exhaustive testing*, whereas for 64-bit floating-point or 64-bit fixed-point math libraries, these proofs can either be interactive [54], [77] or fully automatic [38], [78], [108]. Since our focus is on math libraries for ML, we choose the exhaustive testing approach for our math library, specifically, we 1) run our implementations on all possible inputs, 2) compare the ULP error between each output and the infinite precision real result, and 3) report the maximum observed ULP error as the bound. For step 2, we need the ability to compute math functions to arbitrary degrees of precision – this is offered by the GNU MPFR library [45].

We prove ULP error bounds for bitwidth 16 (Section VI-C) and appropriate input/output scales, s_x and s_y, and choose parameters d, g, and t accordingly to ensure high precision. Note that given a bitwidth ℓ, a proof via exhaustive testing requires 2^{ℓ} tests. For exponential, we set $d = 8$ and prove that $\forall s_x, s_y \in [8, 14]$, the maximum ULP error is 3. For sigmoid and tanh, we set $d = 8$, $g = \lceil \frac{s_y - 2}{2} \rceil$ and $t = 0$, and prove that $\forall s_x, s_y \in [8, 14]$ the maximum ULP error is 3 for sigmoid and 4 for tanh. For reciprocal square root, we choose inputs $x \geqslant \epsilon$ where $\epsilon = 0.1$, and set $g = \lceil \frac{s_y}{2} \rceil$ and $t = 1$. We prove that $\forall s_x, s_y \in [4, 13]$, the maximum ULP error is 4.

Thus, using exhaustive testing, we prove that our math implementations are precise for chosen parameters and provide standard precision guarantees that are expected from math libraries viz. ULP error < 5; Intel's SVML [4] also provides math implementations with 4 ULP error. We use the same parameter setting described above for the empirical evaluation.

VI. Evaluation

In this section, we empirically compare our protocols for math functions with prior works and describe the results of our ML case studies. The closest work to ours is MiniONN [83], the only prior work on secure inference that has been evaluated on an RNN. MiniONN proposes a recipe to obtain piecewise linear approximations to sigmoid/tanh that are then evaluated using its protocols. Our secure implementations of sigmoid are an order of magnitude better in communication (Table I). Note that no prior work on 2-party secure inference (including MiniONN) provides secure implementations of exponentiation and reciprocal square root; we evaluate them in Table II. General-purpose MPC frameworks like MP-SPDZ [66] also provide semi-honest 2PC implementations of math functions [3] that are compatible with the standard (power-of-2 ring-based) fixed-point representation. However, the communication of our protocols is up to two orders of magnitude lower. Alternatives that use representations such as field-based representations or floating-point also suffer from high communication overheads.

Next, we evaluate our library SiRNN for DNN inference on end-to-end ML models. First, we evaluate SiRNN on models with math functions considered by priors works [83], [102]. Since they evaluate sigmoid and tanh using generic 2PC protocols, SiRNN has an order of magnitude less communication (Table III). Next, we evaluate SiRNN on RNNs for sports training and audio keyword spotting that use GRU cells, which are composed of sigmoid and tanh operators. There are two ways to securely evaluate our math functionalities, with our 2PC protocols and with generic 2PC protocols for mixed

arithmetic and boolean compute [25], [32], [41], [95]. We evaluate both and observe that SIRNN communicates over 500× less data for both the RNNs (Table IV). Finally, we evaluate SIRNN on a recent model architecture that combines CNN operators and RNN operators to find the human heads in images with state-of-the-art accuracy [104]. We provide the first secure implementation for this complex model; its secure implementation requires all the protocols described in this paper including reciprocal square root and takes less than 7 minutes on our evaluation set up:

System Details. We use a set up where the 2 machines are connected via a 377 MBps LAN network with 0.8 ms RTT. Both the machines have commodity hardware with a 4-core 3.7 GHz Xeon processor and 16 GBs of RAM.

Implementation Details. The users of SIRNN express their DNNs as a combination of calls to SIRNN's C++ library functions. These functions include matrix multiplication, convolutions, MBConv blocks, L2 Normalization, batch normalization, broadcasting; pointwise operators like sigmoid, tanh, exponential, reciprocal square root, matrix addition, Hadamard product; comparison-based operators like argmax, maxpool, ReLU, and ReLU6. The last four functions use protocols from [99] and the rest use our building blocks. The library functions take scales as arguments and are templated on the bitwidths. The SIRNN library is implemented using 28K lines of C++. We statically generate 36 LUTs that consume additional 35K LOC.

A. Microbenchmarks

a) Sigmoid: In Table I, we compare our protocol with prior work for generating sigmoid output with 12-bits of precision (i.e., scale 12). We report absolute numbers for time taken and communication for both our protocols and prior work, as well as improvement factor of our protocols in parentheses. We follow this pattern for all the tables in this section. We focus on sigmoid as the numbers for tanh are similar. One sigmoid evaluation with our protocols incurs less than 5KB of communication and produces precise results with at most 3 ULPs error. In ML, sigmoid is usually computed pointwise over all the entries in a tensor. Hence, one needs to compute sigmoid of a large number of instances when dealing with realistic ML benchmarks. Although the communication to compute n sigmoid instances grows linearly with n, empirically we have observed that the time taken or the latency grows sub-linearly with n (columns 2 to 5 of Table I), which helps our implementations to scale well to large tensors (Section VI-C). The cost of rounds amortizes better for large tensors resulting in the sub-linear growth in latency.

As a baseline, we consider the recipe of MiniONN that approximates math functions with piecewise linear approximations and provides protocols to evaluate these splines. More precise approximations require more number of pieces. To get an ULP error below 5, MiniONN needs a 48-way spline which provides poor performance when evaluated securely because of a 70× communication overhead.

Technique	Total Time for #Instances (in sec)				Comm./ Instance (in KB)	Max ULP Err.
	10^2	10^3	10^4	10^5		
Our Work	0.08	0.10	0.25	1.58	4.88	3
MiniONN 48-piece	0.20 (2.5x)	1.94 (19.4x)	18.85 (75x)	182.2 (115x)	341.03 (70x)	4
MiniONN 12-piece	0.06 (0.8x)	0.54 (5.4x)	5.24 (21x)	53.84 (34x)	93.36 (19.1x)	104
Deep-Secure	0.16 (2x)	0.84 (8.4x)	8.1 (32x)	141.3 (89x)	124.65 (25x)	NA
MP-SPDZ Ring Poly	0.75 (9.4x)	1.72 (17.2x)	14.88 (59.5x)	140.6 (89x)	981.11 (201x)	2
MP-SPDZ Ring PL	0.27 (3.4x)	0.28 (2.8x)	1.32 (5.3x)	12.34 (7.8x)	76.42 (15.7x)	266
MP-SPDZ Field Poly	0.91 (11.4x)	1.91 (19.1x)	16.51 (66x)	127 (80x)	228.63 (46.9x)	2
MP-SPDZ Field PL	0.52 (6.5x)	0.47 (4.7x)	1.79 (7.2x)	14.23 (9x)	27.52 (5.6x)	266

TABLE I: Comparison with prior works on sigmoid with varying number of instances.

For the RNN benchmark that MiniONN considers (Section VI-B), the precision offered by the 48-piece spline is an overkill and a 12-piece spline suffices to maintain the cross entropy loss. Although this 12-piece spline is more efficient than 48-piece spline, its performance is still much worse than our protocols and incurs a 19× communication overhead. Furthermore, this 12-piece spline incurs an error of 104 ULPs. Hence, our implementations are superior in both precision and performance. While a 12-piece spline suffices for this benchmark, MiniONN remarks that other benchmarks need splines with more number of pieces that are even more expensive to compute. Because our implementations are guaranteed to be numerically precise, they can be used as-is with no loss in model accuracy (Section VI-C).

DeepSecure [102] uses garbled circuits (GC) to evaluate DNNs that use sigmoid and tanh activations. We checked with the authors of DeepSecure and the circuits for math functions are not available. Hence, we cannot compute the ULP errors of their implementations. However, DeepSecure reports the number of non-XOR gates that can be used for performance estimates. We used state-of-the-art for GC implementation, i.e., EMP-Toolkit [1], [52], [53], to obtain these performance estimates that are better than the performance reported by DeepSecure. The communication of our protocols is 25× lower (4^{th} row of Table I).

MP-SPDZ [66], a general-purpose MPC framework, provides 2 baseline sigmoid implementations for 2PC [3]: Poly-based, which uses a range reduction and Taylor series polynomials to compute exponential followed by division, and PL-based, which is a built-in piecewise linear spline. The former implementation incurs error comparable to us but communicates 201× more, while the latter is more than an order of magnitude inferior in precision and communication (5^{th} and 6^{th} row of Table I).

While we focus on power-of-2 rings, there are other works on secure implementations of sigmoid that use field-based or

Technique	Total Time for #Instances (in sec)				Comm./ Instance (in KB)	Max ULP Error
	10^2	10^3	10^4	10^5		
Exponentiation						
Our Work	0.03	0.04	0.15	1.00	2.12	3
MP-SPDZ	0.34 (11.3x)	0.56 (14x)	3.90 (26x)	35.95 (35.9x)	254.95 (120x)	2
Reciprocal Square Root						
Our Work	0.13	0.13	0.30	1.84	6	4
MP-SPDZ	0.94 (7.2x)	3.90 (30x)	35.87 (120x)	338.9 (184x)	2535 (423x)	8

TABLE II: Comparison with (power-of-2) ring-based MP-SPDZ protocols with varying number of instances.

floating point representations. Field-based protocols perform poorly for non-linear computations like truncation and comparisons, which are abundant in fixed-point representations of DNNs [64], [90], [99]. Similarly, it is well-known that the protocols over floating-point are much slower than fixed-point [29], [73]. Nonetheless, for completeness, we compare against the state-of-the-art field-based implementations in MP-SPDZ [3], [66] and they perform worse (7^{th} and 8^{th} rows of Table I). We also compare with floating-point implementations of math functions provided by ABY [40] and EMP-Toolkit [1]; our protocols are at least $90\times$ better in communication per instance and $97\times$ better in runtime (for 10^5 instances).

Finally, SecureML [92] and ABY2.0 [95] use a 3-piece linear spline to approximate sigmoid. This simple implementation has a whopping error of 1547 ULPs and tanks the accuracy of our RNN benchmarks. For instance, it leads to a tremendous drop in accuracy of the Google-30 network from 84.4% (with our sigmoid implementation) to 60.95%. The insufficiency of this approximation has also been noted by [83] where it caused the cross-entropy loss to diverge to infinity. Hence, this crude approximation is usable only in restricted contexts and is unsuitable for generic math libraries, which is our aim here.

b) Exponential and reciprocal square-root: Table II shows the comparison of our exponentiation and reciprocal square-root protocols with power-of-2 ring based protocols in MP-SPDZ framework (for scale 12). It has native support for exponentiation. We implement reciprocal square root in MP-SPDZ by calling its built-in functions for square root and reciprocal. As the table shows, our protocols are orders of magnitude better, both in terms of time-taken and communication, and provide better or comparable ULP errors.

B. Prior DNNs

In Table III, we evaluate our protocols on benchmarks with math functions from MiniONN [83] and DeepSecure [102]. MiniONN evaluated an LSTM for text data which has 2 LSTM layers each with 800 instances of sigmoid and 200 instances of tanh. Our protocols incur an order of magnitude less communication for these instances. We consider the largest benchmark of DeepSecure, B4, with 2 tanh layers of 2000 and 500 instances, which classifies sensor data into 19 different physical activities. To estimate the time taken by

Inference Benchmark	Runtime (in sec)		Comm.	
	Prior	Our Work	Prior	Our Work
MiniONN LSTM	1.1 (2.2x)	0.48	182 MB (19.5x)	9.32 MB
DeepSecure B4	465 (87x)	5.3	83.7 GB (43x)	1.94 GB

TABLE III: Comparison with benchmarks from MiniONN [83] and DeepSecure [102].

Benchmark	Batch	Runtime (sec)		Comm.	
		[41]	SIRNN	[41]	SIRNN
Industrial-72	1	68.33 (18x)	3.7	11.84 GB (510x)	23.8 MB
	128	8746* (661x)	13.2	1.47 TB* (1451x)	1.04 GB
Google-30	1	3337 (67x)	49.6	259 GB (574x)	0.45 GB
	128	$4.3\text{x}10^5$* (3050x)	140	32.38 TB* (1316x)	25.2 GB
Heads	1	NA	409.7	NA	85.5 GB

*extrapolated, the run could not be completed due to TB comm.

TABLE IV: Secure inference on DNNs using SIRNN and [41].

DeepSecure on our setup, we ran a circuit with the same non-XOR complexity as B4 using EMP-Toolkit [1] (similar to our microbenchmarks) that provides better performance than the communication and latency in [102]. Our protocols have $87\times$ lower latency and $43\times$ lower communication.

C. Case studies

We demonstrate the applicability of secure inference to three new domains that no prior work has considered before: RNNs applied to time series sensor data, RNNs applied to speech data, and combining CNNs and RNNs to identify human heads in images. The feasibility of our case studies crucially relies on our efficient protocols for math functions. Our first case study is an industrial model (Industrial [72]) which uses an RNN with GRU cells to provide feedback on the quality of shots in a bat-and-ball game from the data obtained from sensors deployed on the bat. Second, we evaluate an RNN (Google-30 [74]) for keyword spotting in the standard Google-30 [112] dataset that identifies simple commands, digits, and directions from speech data obtained from thousands of people. Third, the head detection model (Heads [104]) combines CNNs and RNNs for the best accuracy on the SCUT Head dataset [96]. It uses inverted residual blocks, or MBConv blocks [105], for efficient convolutions. Instead of simple pooling operators like maxpool or average pool, it uses RNN-based pooling that provides high accuracy. We summarize the input fixed-point code of these benchmarks below. These fixed-point C++ programs were automatically generated from high-level ML models by [72] (a compiler for embedded devices) and linked with SIRNN. All of the benchmarks use a mixture of variables with bitwidth 8, 16, and 32 with 16 being the bitwidth used for input and output of the math functions.

- *Industrial-72*: It contains 7 sigmoid and 7 tanh layers, with 64 instances each. While sigmoid uses the input

scale 8 and output scale 14, for tanh both scales are 8.

- *Google-30*: It contains 99 sigmoid and 99 tanh layers, with 100 instances each. While sigmoid uses the input scale 6 and output scale 14, for tanh both scales are 6.
- *Heads*: It contains 128 sigmoid and 128 tanh layers, with 18096 instances each. While sigmoid uses the input scale 11 and output scale 14, for tanh both scales are 11. Additionally, the benchmark contains 8 sigmoid and 8 tanh layers, with 72384 instances each. For these layers, sigmoid uses the input scale 13 and output scale 14, and for tanh both scales are 13. Finally, it also contains 3 L2-Normalise layers that have 1200, 1200 and 300 reciprocal square-root operations. The layers have input scales 12, 10 and 12 and output scales 11, 9 and 11, respectively.

Note that the Heads model makes about 3 million calls to sigmoid/tanh, which is three orders of magnitude larger than the number of calls to these functions in the benchmarks used by prior work (Section VI-B).

In Table IV, we present the latency and communication required by SIRNN on above benchmarks. Using our protocols, Industrial takes 4 seconds, Google-30 takes under a minute, and Heads takes less than 7 minutes. The time per inference can be further improved by *batching* multiple predictions. For a batch size of 128, the amortized time per inference of Industrial is 0.1s and of Google-30 is 1.1s! The savings in batching come from amortizing the networking cost by packing data from multiple inference queries. Owing to the high numerical precision of our math functionalities (Section V-D), SIRNN either matches or exceeds the model accuracy of the provided fixed-point ML model. In Heads, about half the time is spent in math operations and the rest of the time is spent in matrix multiplications, convolutions, and Hadamard products. The good performance on end-to-end benchmarks is a result of co-designing precise math functionalities and efficient protocols.

Next, we perform an ablation study. In particular, the fixed-point code with our math functionalities can be run with other protocols. However, prior work on secure inference don't support juggling between different bitwidths that our math functionalities require. Hence, for running these functionalities with any prior protocol, we need to use an appropriately large uniform bitwidth. We evaluate our benchmarks with ABY [41] using the necessary bitwidth of 64 as a baseline in Table IV. ABY [41] provides general purpose state-of-the-art 2PC protocols that have been used by recent work on secure inference [25], [32], [83], [92]. We have added a new code generator to [72] that generates EzPC [32] code which is then automatically translated to ABY code. Other generic protocols that have suitable frontends [60], [82], [88], [109], like garbled circuits, are several orders of magnitude slower than ABY [32], [92]: ML inference involves many multiplications that are very expensive with garbled circuits. SIRNN is over $500\times$ better than ABY in communication and more than an order of magnitude faster in runtime. Without our protocols, it takes almost an hour to run Google-30. This situation is further exacerbated on bigger models and running the Heads model with ABY is intractable because it requires hundreds of terabytes of communication. With batching, the performance differences are stark: SIRNN is three orders of magnitude better in latency and communication compared to the ABY baseline.

VII. Other Related work

Prior 2PC works that use high degree polynomials for approximating math functions [9], [34], [57], [68] need degree 7 or higher to maintain accuracy. In the course of this work, we have observed that evaluating polynomials with degree 3 or higher with 2PC is much more expensive than the LUT-based implementations of Section V. Some prior works on secure inference implement math functions with ad hoc approximations that can lose model accuracy: e.g. SecureML [92] and ABY2.0 [95] use a crude 3-piece linear approximation, Ball et al. [13] replace tanh with the signum function, and Glyph [85] and Nandakumar et al. [93] use tables of approximate results Most recent works on secure inference limit their evaluation to benchmarks that don't use math functions [17], [39], [44], [47], [64], [90], [99]. Prior 2PC works that use floating-point representations (instead of fixed-point representations) have much higher performance overheads [1], [6], [7], [12], [33], [40], [46], [66], [84]

Other relevant works that need additional parties to ensure security such as 3PC with honest majority or 2PC with trusted dealer include [8], [9], [28]–[31], [43], [86], Chameleon [101], CrypTen [69], TF-Encrypted [2], CrypTFlow [73], PySyft [103], ABY^3 [91], SecureQ8 [37], and Sharemind [65], [67], [71], [75], [97]. Some of these works have considered approximations to math functions and, similar to 2PC works, they either use polynomial-based approximations (e.g. [9], [71], [86]) or work over floating-point (e.g. [8], [29], [30], [65], [67], [75], [97]). Kerik et al. [67] also consider building blocks such as extension, truncate-and-reduce, and multiplication of non-uniform bitwidths in the 3PC context. In terms of representations, while floating-point and fixed-point representations are most common, [43] proposed the new representations of golden-section and logarithmic numbers and evaluated using 3PC protocols.

Recent works on silent-OT [22], [114] provide OT extensions with much lower communication than IKNP-style extensions [62], at the cost of higher computational overhead. Since our protocols make use of OTs in a black-box manner, silent-OT can be used to obtain lower communication. However, in our setting, when the IKNP-OT instances are computed by multiple threads and are "load-balanced" (i.e., each party plays the role of the sender in half the OT instances and as the receiver in the other half), we empirically observe that IKNP-style extensions are more performant than silent-OT in our LAN evaluation environment. Hence, SIRNN uses IKNP-style OT extensions in Section VI.

Acknowledgement

We thank Pratik Bhatu, Aayan Kumar, and Aditya Kusupathi for their help with the implementation and the evaluation.

REFERENCES

[1] "EMP-toolkit: Efficient MultiParty computation toolkit," https://github.com/emp-toolkit, 2016.

[2] "TF-Encrypted: A Framework for Encrypted Machine Learning in TensorFlow," https://github.com/tf-encrypted/tf-encrypted, 2018.

[3] "Multi-Protocol SPDZ: Versatile framework for multi-party computation," 2019. [Online]. Available: https://github.com/data61/MP-SPDZ

[4] "Intel SVML," https://software.intel.com/content/www/us/en/develop/documentation/mkl-vmperfdata/top.html, 2020.

[5] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: Two-Party Secure Neural Network Training and Prediction," in *CCS 2019*.

[6] M. Aliasgari and M. Blanton, "Secure computation of hidden markov models," in *SECRYPT*, 2013.

[7] M. Aliasgari, M. Blanton, and F. Bayatbabolghani, "Secure computation of hidden markov models and secure floating-point arithmetic in the malicious model," *Int. J. Inf. Sec.*, 2017.

[8] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele, "Secure computation on floating point numbers," in *NDSS*, 2013.

[9] A. Aly and N. P. Smart, "Benchmarking privacy preserving scientific operations," in *ACNS*, 2019.

[10] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "Ramparts: A programmer-friendly system for building homomorphic encryption applications," in *WAHC 2019*.

[11] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, "More efficient oblivious transfer and extensions for faster secure computation," in *CCS 2013*.

[12] S. Bai, G. Yang, J. Shi, G. Liu, and Z. Min, "Privacy-preserving oriented floating-point number fully homomorphic encryption scheme," *Secur. Commun. Networks 2018*.

[13] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski, "Garbled Neural Networks are Practical," *ePrint 2019/338*.

[14] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe, "Automatic conversion of floating point matlab programs into fixed point fpga based hardware design," in *FCCM 2003*.

[15] D. Beaver, "Efficient Multiparty Protocols Using Circuit Randomization," in *CRYPTO 1991*.

[16] G. R. Blakley, "Safeguarding cryptographic keys," in *Managing Requirements Knowledge, International Workshop on*, 1979.

[17] F. Boemer, R. Cammarota, D. Demmler, T. Schneider, and H. Yalame, "MP2ML: a mixed-protocol machine learning framework for private inference," in *ARES 2020*.

[18] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data," in *WAHC 2019*.

[19] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data," in *CF 2019*.

[20] C. Boura, N. Gama, and M. Georgieva, "Chimera: a unified framework for B/FV, TFHE and HEAAN fully homomorphic encryption and predictions for deep learning," *ePrint 2018/758*.

[21] 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," *ePrint 2020/1392*.

[22] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, "Efficient two-round OT extension and silent non-interactive secure computation," in *CCS*. ACM, 2019, pp. 291–308.

[23] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko, "MOTION - A Framework for Mixed-Protocol Multi-Party Computation," *ePrint 2020/1137*.

[24] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *HPCA 1999*.

[25] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider, "HyCC: Compilation of Hybrid Protocols for Practical Secure Computation," in *CCS 2018*.

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

[27] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *SCC 2015*.

[28] O. Catrina, "Round-efficient protocols for secure multiparty fixed-point arithmetic," in *COMM 2018*.

[29] ——, "Efficient Secure Floating-point Arithmetic using Shamir Secret Sharing," in *ICETE (2)*, 2019.

[30] ——, "Evaluation of floating-point arithmetic protocols based on shamir secret sharing," in *ICETE (Selected Papers)*, 2019.

[31] O. Catrina and A. Saxena, "Secure computation with fixed-point numbers," in *Financial Cryptography*, 2010.

[32] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning," in *IEEE EuroS&P 2019*.

[33] Y. Chang and C. Lu, "Oblivious polynomial evaluation and oblivious neural learning," in *ASIACRYPT*, 2001.

[34] V. Chen, V. Pastro, and M. Raykova, "Secure Computation for Machine Learning With SPDZ," in *PPML 2018, NeurIPS 2018 Workshop*.

[35] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *ASIACRYPT 2016*.

[36] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," in *SSST-8, 2014*.

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

[38] E. Darulova and V. Kuncak, "Sound compilation of reals," in *POPL 2014*.

[39] R. Dathathri, O. Saarikivi, H. Chen, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing," in *PLDI 2019*.

[40] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni, "Automated synthesis of optimized circuits for secure computation," in *CCS 2015*.

[41] D. Demmler, T. Schneider, and M. Zohner, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *NDSS 2015*.

[42] G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, "Pushing the Communication Barrier in Secure Computation using Lookup Tables," in *NDSS 2017*.

[43] V. Dimitrov, L. Kerik, T. Krips, J. Randmets, and J. Willemson, "Alternative implementations of secure real numbers," in *CCS 2016*.

[44] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits," in *CRYPTO 2020*.

[45] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, 2007.

[46] M. Franz and S. Katzenbeisser, "Processing encrypted floating point signals," in *MM&Sec 2011*.

[47] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *ICML 2016*.

[48] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, 1991.

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

[50] R. E. Goldschmidt, "Applications of division by convergence," M.S. thesis, MIT, 1964.

[51] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling KB-Sized Machine Learning Models to Tiny IoT Devices," in *PLDI 2019*.

[52] C. Guo, J. Katz, X. Wang, C. Weng, and Y. Yu, "Better concrete security for half-gates garbling (in the multi-instance setting)," in *CRYPTO (2)*, 2020.

[53] C. Guo, J. Katz, X. Wang, and Y. Yu, "Efficient and secure multiparty computation from fixed-key block ciphers," in *IEEE Symposium on Security and Privacy*, 2020.

[54] J. Harrison, "A machine-checked theory of floating point arithmetic," in *Theorem Proving in Higher Order Logics*, 1999.

[55] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning (2nd Edition)*, 2009.

[56] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *CVPR 2016*.

[57] B. Hemenway, S. Lu, R. Ostrovsky, and W. W. IV, "High-precision secure computation of satellite collision probabilities," in *SCN 2016*.

[58] E. Hesamifard, H. Takabi, and M. Ghasemi, "CryptoDL: Deep Neural Networks over Encrypted Data," *CoRR 2017*.

[59] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, 1997.

[60] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *CCS 2012*.

[61] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *CVPR 2017*.

[62] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending Oblivious Transfers Efficiently," in *CRYPTO 2003*.

[63] M. Ito, N. Takagi, and S. Yajima, "Efficient Initial Approximation for Multiplicative Division and Square Root by a Multiplication with Operand Modification," *IEEE Transactions on Computers*, 1997.

[64] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in *USENIX Security 2018*.

[65] L. Kamm and J. Willemson, "Secure floating point arithmetic and private satellite collision analysis," *Int. J. Inf. Sec.*, 2015.

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

[67] L. Kerik, P. Laud, and J. Randmets, "Optimizing MPC for robust and scalable integer and floating-point arithmetic," in *Financial Cryptography Workshops 2016*.

[68] D. Kim, Y. Son, D. Kim, A. Kim, S. Hong, and J. H. Cheon, "Privacy-preserving approximate GWAS computation based on homomorphic encryption," *ePrint 2019/152*.

[69] B. Knott, S. Venkataraman, A. Hannun, S. Sengupta, M. Ibrahim, and L. van der Maaten, "CrypTen: Secure multi-party computation meets machine learning," in *Workshop on Privacy Preserving Machine Learning, December 11, 2020*.

[70] V. Kolesnikov and R. Kumaresan, "Improved OT Extension for Transferring Short Secrets," in *CRYPTO 2013*.

[71] T. Krips and J. Willemson, "Hybrid model of fixed and floating point numbers in secure multiparty computations," in *ISC*, 2014.

[72] A. Kumar, V. Seshadri, and R. Sharma, "Shiftry: RNN Inference in 2KB of RAM," in *OOPSLA*, 2020.

[73] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "CrypTFlow: Secure TensorFlow Inference," in *IEEE S&P 2020*.

[74] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network," in *NeurIPS 2018*.

[75] P. Laud and J. Randmets, "A domain-specific language for low-level secure multiparty computation protocols," in *CCS 2015*.

[76] S. Laur, H. Lipmaa, and T. Mielikäinen, "Cryptographically private support vector machines," in *SIGKDD 2006*.

[77] J. Le Maire, N. Brunie, F. De Dinechin, and J. Muller, "Computing floating-point logarithms with fixed-point operations," in *IEEE ARITH 2016*.

[78] W. Lee, R. Sharma, and A. Aiken, "On automatically proving the correctness of math.h implementations," in *POPL 2018*.

[79] ——, "Verifying bit-manipulations of floating-point," in *PLDI 2016*.

[80] K.-P. Lin and M.-S. Chen, "Privacy-preserving outsourcing support vector machines with random transformation," in *SIGKDD 2010*.

[81] Y. Lindell, *How to Simulate It – A Tutorial on the Simulation Proof Technique*, 2017.

[82] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A Programming Framework for Secure Computation," in *IEEE S&P 2015*.

[83] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious Neural Network Predictions via MiniONN Transformations," in *CCS 2017*.

[84] Y. Liu, Y. Chiang, T. Hsu, C. Liau, and D. Wang, "Floating point arithmetic protocols for constructing secure data analysis application," in *KES*, 2013.

[85] Q. Lou, B. Feng, G. Charles Fox, and L. Jiang, "Glyph: Fast and accurately training deep neural networks on encrypted data," *to appear in NeurIPS 2020*.

[86] W.-j. Lu, Y. Fang, Z. Huang, C. Hong, C. Chen, H. Qu, Y. Zhou, and K. Ren, "Faster secure multiparty computation of adaptive gradient descent," *to appear in PPML 2020, NeurIPS 2020 Workshop*.

[87] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren, "EPIC: Efficient Private Image Classification (or: Learning from the Masters)," in *CT-RSA 2019*.

[88] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - Secure Two-Party Computation System," in *USENIX Security 2004*.

[89] D. Menard, D. Chillet, F. Charot, and O. Sentieys, "Automatic floating-point to fixed-point conversion for dsp code generation," in *CASES 2002*.

[90] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A Cryptographic Inference Service for Neural Networks," in *USENIX Security 2020*.

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

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

[93] K. Nandakumar, N. K. Ratha, S. Pankanti, and S. Halevi, "Towards deep neural network training on encrypted data," in *CVPR Workshops*, 2019.

[94] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Precision and error analysis of matlab applications during automated hardware synthesis for FPGAs," in *DATE 2001*.

[95] A. Patra, T. Schneider, A. Suresh, and H. Yalame, "ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation," *to appear in USENIX Security 2021*.

[96] D. Peng, Z. Sun, Z. Chen, Z. Cai, L. Xie, and L. Jin, "Detecting heads using feature refine net and cascaded multi-scale architecture," *arXiv 2018*.

[97] P. Pullonen and S. Siim, "Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations," in *Financial Cryptography Workshops*, 2015.

[98] Y. Rahulamathavan, R. C. . Phan, S. Veluru, K. Cumanan, and M. Rajarajan, "Privacy-preserving multi-class support vector machine for outsourcing the data classification in cloud," *TDSC 2014*.

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

[100] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: XNOR-based Oblivious Deep Neural Network Inference," in *USENIX Security 2019*.

[101] 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*.

[102] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable Provably-Secure Deep Learning," in *DAC 2018*.

[103] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, "A generic framework for privacy preserving deep learning," *CoRR*, 2018.

[104] O. Saha, A. Kusupati, H. V. Simhadri, M. Varma, and P. Jain, "RNNPool: Efficient Non-linear Pooling for RAM Constrained Inference," *to appear in NeurIPS 2020*.

[105] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR 2018*.

[106] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Optimization of Floating-Point Programs with Tunable Precision," in *PLDI 2014*.

[107] A. Shamir, "How to share a secret," *Commun. ACM*, 1979.

[108] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," *ACM Trans. Program. Lang. Syst.*, 2018.

[109] E. M. Songhori, M. S. Riazi, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "ARM2GC: Succinct garbled processor for secure computation," *arXiv 2019*.

[110] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-Party Secure Computation for Neural Network Training," *PoPETs 2019*.

[111] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *Intel Math Kernel Library*, 2014.

[112] P. Warden, "Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition," *arXiv 2018*.

[113] W.-F. Wong and E. Goto, "Fast evaluation of the elementary functions in single precision," *IEEE Transactions on Computers 1995*.

[114] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated OT with small communication," in *CCS*. ACM, 2020, pp. 1607–1626.

[115] A. C. Yao, "How to Generate and Exchange Secrets (Extended Abstract)," in *FOCS 1986*.

APPENDIX

A. Optimized Protocol for $\mathcal{F}^{\ell}_{\mathsf{MUX}}$

In this section, we present an optimized protocol for $\mathcal{F}^{\ell}_{\mathsf{MUX}}$ which utilizes COT and builds over the protocol used in [99]. Our optimization relies on the following observation: consider $x \in \mathbb{Z}_2$ with shares $\langle x \rangle^B = (x_0, x_1)$ and $y \in \mathbb{Z}_L$ with shares $\langle y \rangle^{\ell} = (y_0, y_1)$, then the following holds:

$$\begin{aligned} x *_{\ell} y &= (x_0 \oplus x_1) *_{\ell} (y_0 + y_1) \\ &= (x_0 + x_1 - 2x_0 *_{\ell} x_1) *_{\ell} (y_0 + y_1) \\ &= x_0 *_{\ell} y_0 + x_1 *_{\ell} (y_0 - 2x_0 *_{\ell} y_0) \\ &\quad + x_1 *_{\ell} y_1 + x_0 *_{\ell} (y_1 - 2x_1 *_{\ell} y_1) \end{aligned}$$

In the above, the terms $x_0 *_{\ell} y_0$ and $x_1 *_{\ell} y_1$ can be locally computed by P_0 and P_1, respectively, while for the other two terms, we use $\binom{2}{1}$-COT_{ℓ} protocol. In particular, to calculate shares of $x_1 *_{\ell} (y_0 - 2x_0 *_{\ell} y_0)$ term, P_0 acts as the sender with correlation $(y_0 - 2x_0 *_{\ell} y_0)$ and P_1 acts as the receiver with choice bit x_1; similarly the term can be computed with the sender and receiver roles reversed. Note that both the COTs can be done in parallel giving us a 2-round solution which communicates 2ℓ less bits than prior approach from [99] that used 2 instances of $\binom{2}{1}$-OT_{ℓ}.

B. Wrap and All Ones

Recall that the functionality $\mathcal{F}^{\ell}_{\mathsf{Wrap\&All1s}}(x, y)$ outputs $(\langle w \rangle^B || \langle e \rangle^B)$ such that $w = \mathsf{wrap}(x, y, L)$ and $e = \mathbf{1}\{(x + y \bmod L) = L - 1\}$. Consider the ℓ-bit functionality $\mathcal{F}^{\ell}_{\mathsf{Eq}}(x, y)$ that returns $\langle e \rangle^B$ such that $e = \mathbf{1}\{x = y\}$. Then, $\mathcal{F}^{\ell}_{\mathsf{Wrap\&All1s}}(x, y) = \mathcal{F}^{\ell}_{\mathsf{Mill}}(L - 1 - x, y) || \mathcal{F}^{\ell}_{\mathsf{Eq}}(L - 1 - x, y)$, that is, millionaires' and equality on the same inputs. Now, to construct an efficient protocol for $\mathcal{F}^{\ell}_{\mathsf{Mill}}$, CrypTFlow2 [99] used the following recurrence relations: Let $x = (x_1 || x_0)$ and $y = (y_1 || y_0)$ such that $x_i, y_i \in \{0,1\}^{\ell/2}$ for $i \in \{0, 1\}$. Then,

$$\begin{aligned} \mathbf{1}\{x < y\} &= \mathbf{1}\{x_1 < y_1\} \oplus (\mathbf{1}\{x_1 = y_1\} \wedge \mathbf{1}\{x_0 < y_0\}) \\ \mathbf{1}\{x = y\} &= \mathbf{1}\{x_1 = y_1\} \wedge \mathbf{1}\{x_0 = y_0\} \end{aligned}$$

That is, they reduce the millionaires' on ℓ-bit strings to millionaires' and equalities on smaller strings. While they computed millionaires' instances on all nodes, they skipped a small number of equality computations that were not used, e.g. the root note. For $\mathcal{F}^{\ell}_{\mathsf{Wrap\&All1s}}$, we compute millionaires' and equality on all notes and this marginally increases the cost over the protocol for $\mathcal{F}^{\ell}_{\mathsf{Mill}}$. Nonetheless, the communication cost of $\mathcal{F}^{\ell}_{\mathsf{Wrap\&All1s}}$ is at most $\lambda\ell + 14\ell$.

C. Truncation

1) Proof for Lemma 1: For $b \in \{0, 1\}$, let $x_b = \langle x \rangle^{\ell}_b$. Over $\mathbb{Z}$, we can write $x_b = u_b \cdot 2^s + v_b$ and have the following:

$$\begin{aligned} x_0 + x_1 &= (v_0 + v_1) + 2^s(u_0 + u_1) \\ &= (v_0 + v_1 - c \cdot 2^s) + 2^s(u_0 + u_1 - d \cdot 2^{\ell - s}) \\ &\quad + c \cdot 2^s + d \cdot L \\ &= v' + 2^s(u' + c) + d \cdot L \end{aligned}$$

Algorithm 4 Cross Term Multiplication, $\Pi^{m,n}_{\mathsf{CrossTerm}}$:

Input: P_0 holds $x \in \mathbb{Z}_M$ and P_1 holds $y \in \mathbb{Z}_N$, where $m \leqslant n$.
Output: P_0 & P_1 get $\langle z \rangle^{\ell}_b$, where $z = x *_{\ell} y$ and $\ell = m + n$.

1: P_0 parses x as an m-bit string $x = x_{m-1} || \cdots || x_0$, where $x_i \in \{0, 1\}$.
2: **for** $i = \{0, \ldots, m-1\}$ **do**
3: P_0 & P_1 invoke $\binom{2}{1}$-$\mathsf{COT}_{\ell-i}$, where P_0 is the sender with input x_i and P_1 is the receiver with input y, and learn $\langle t_i \rangle^{\ell-i}$.
4: **end for**
5: For $b \in \{0, 1\}$, P_b sets $\langle z \rangle^{\ell}_b = \sum_{i=0}^{m-1} 2^i \cdot \langle t_i \rangle^{\ell-i}_b$.

Let $w' = \mathbf{1}\{u' + c > 2^{\ell - s} - 1\}$. Then

$$x_0 + x_1 = v' + 2^s(u' + c - w' \cdot 2^{\ell - s}) + L \cdot (d + w') \quad (3)$$

When $d = 1$, then $e = 0$ and $u' = u_0 + u_1 - 2^{\ell - s}$. Since $u_0, u_1 \leqslant 2^{\ell - s} - 1$, we have that $u' \leqslant 2^{\ell - s} - 2$. Therefore, $w' = 0$ (because $c \in \{0, 1\}$). On the other hand when $d = 0$, $u' = u_0 + u_1 \leqslant 2^{\ell - s} - 1$. Therefore, $w' = 1$ when $u' = 2^{\ell - s} - 1$ (i.e., $e = 1$) as well as $c = 1$, and 0 otherwise. Since at most one of d and w' is 1 in any given case, we can rewrite Equation 3 as:

$$x_0 + x_1 = v' + 2^s(u' + c - w' \cdot 2^{\ell - s}) + L \cdot (d \oplus (c \wedge e))$$

Since $v' < 2^s$ and $u' + c - w' \cdot 2^{\ell - s} < 2^{\ell - s}$, $w = d \oplus (c \wedge e)$.

2) Division by power-of-2: We can write $\mathsf{DivPow2}(x, s) = (x \gg_A s) + m_x \wedge c$, where $m_x = \mathbf{1}\{x \geqslant 2^{\ell - 1}\}$ is the MSB of x and $c = \mathbf{1}\{x \bmod 2^s \neq 0\}$. In this equation, m_x can be computed with a call to $\mathcal{F}^{\ell-1}_{\mathsf{Mill}}$ using the integer DReLU protocol from [99] and c can be computed with an equality check on s-bit inputs. We get $m_x \wedge c$ in ℓ-bits with a call each to $\mathcal{F}_{\mathsf{AND}}$ and $\mathcal{F}^{\ell}_{\mathsf{B2A}}$, and then a final call to $\mathcal{F}^{\ell,s}_{\mathsf{ARS}}$ gives us $\mathsf{DivPow2}(x, s)$. Since we have already computed the MSB of x, we employ the MSB-to-wrap optimization (Section IV-E) here to minimize the cost of $\mathcal{F}^{\ell,s}_{\mathsf{ARS}}$. The exact cost expression for computing DivPow2 is given in Table V.

D. Multiplication

Here, we formally describe our protocols for cross term multiplication $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$ and matrix multiplication.

1) Cross Term Multiplication, $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$: Our protocol for $\mathcal{F}^{m,n}_{\mathsf{CrossTerm}}$ uses COT similar to prior works [41], [92], [99], but unlike prior works, we support operands of different bitlengths. We present our protocol in Algorithm 4 for the $m \leqslant n$ case. When $m > n$, we simply reverse the roles of the parties in our protocol so that only n COTs are performed. Correctness of this protocol follows similarly to the prior works.

2) Matrix Multiplication: Before we look at matrix multiplication, we first set some notation starting with operator $\boxtimes_{\ell} : \mathbb{Z}^{d_1 \times d_2} \times \mathbb{Z}^{d_2 \times d_3} \to \mathbb{Z}_L^{d_1 \times d_3}$, which does a matrix multiplication between two input matrices X and Y such that $X \boxtimes_{\ell} Y = X \times Y \bmod L$. Similarly to the $*_{\ell}$ notation, when one of the matrices has elements over ring $\mathbb{Z}_M$, we use the

lossless typecast operator ζ_m to map all elements of that matrix to $\mathbb{Z}$. All the *single-input* functionalities we consider naturally extend to matrices, where the functionality is independently applied to all elements of the input matrix to output a matrix of the same dimensions. The shares of a matrix $X \in \mathbb{Z}_M^{d_1 \times d_2}$ are denoted by $\langle X \rangle^m$, where $\langle X \rangle^m = \{\langle X[i,j] \rangle^m\}_{i \in [d_1], j \in [d_2]}$, and the shares of its transpose are denoted by $\langle X^T \rangle^m$.

Now, consider the matrix multiplication functionality $\mathcal{F}_{\mathsf{UMatMul}}^{m,n,d_1,d_2,d_3}$ that takes as input $\langle X \rangle^m \in \mathbb{Z}_M^{d_1 \times d_2}$ and $\langle Y \rangle^n \in \mathbb{Z}_N^{d_2 \times d_3}$ and outputs $\langle Z \rangle^\ell \in \mathbb{Z}_L^{d_1 \times d_3}$ such that $\ell = m + n + \lceil \log d_2 \rceil$ and $Z = X \boxtimes_\ell Y$. As described in Section IV-C, we need the additional $e = \lceil \log d_2 \rceil$ bits to prevent integer overflow due to additions. When $m \leqslant n$, we extend the input matrix $\langle Y \rangle^n$ to get $\langle Y' \rangle^{n'}$ for $n' = n + e$. Then, Equation 2 generalizes to matrices as follows:

$X \boxtimes_\ell Y' = X_0 \boxtimes_\ell Y_0' + X_1 \boxtimes_\ell Y_1' + X_0 \boxtimes_\ell Y_1' + X_1 \boxtimes_\ell Y_0' - 2^{n'} *_\ell (X \boxtimes_m W_{Y'}) - M *_\ell (W_X \boxtimes_{n'} Y')$, where $W_X = \mathsf{wrap}(X_0, X_1, M)$ and $W_{Y'} = \mathsf{wrap}(Y_0', Y_1', 2^{n'})$.

Similar to $\mathcal{F}_{\mathsf{CrossTerm}}^{m,n'}$, we define a functionality $\mathcal{F}_{\mathsf{MatCrossTerm}}^{m,n',d_1,d_2,d_3}$ for matrices to compute the cross-terms $X_0 \boxtimes_\ell Y_1'$ and $X_1 \boxtimes_\ell Y_0'$. This functionality can be realized naively by making $d_1 d_2 d_3$ independent calls to $\Pi_{\mathsf{CrossTerm}}^{m,n'}$. Instead, we can do much better by observing that in a matrix multiplication, each element of X is multiplied with d_3 elements of Y. Thus, rather than doing d_3 independent COTs on $\ell - i$ bit-strings in Step 3 of $\Pi_{\mathsf{CrossTerm}}^{m,n'}$, we can perform a single COT on $d_3 \cdot (\ell - i)$ bit-strings (while respecting the independent correlations). This method of batching COTs was also used in prior works on secure inference [92], [99], and it leads to an overall communication of $d_1 d_2 (m\lambda + (mn' + m^2/2 + m/2) d_3)$ bits.

Note that $\langle W_X \rangle^B$ and $\langle W_{Y'} \rangle^B$ can be computed by making $d_1 d_2$ calls to $\mathcal{F}_{\mathsf{Wrap}}^m$ and $d_2 d_3$ calls to $\mathcal{F}_{\mathsf{Wrap}}^{n'}$, respectively. Since the terms $X_i \boxtimes_\ell Y_i'$ can be computed locally, the only terms left to compute are $X \boxtimes_m W_{Y'}$ and $W_X \boxtimes_{n'} Y'$. They can be computed using the following functionality $\mathcal{F}_{\mathsf{BitMatMul}}^{\ell,d_1,d_2,d_3}$ that takes a bit-matrix $\langle W \rangle^B \in \{0,1\}^{d_1 \times d_2}$ and a matrix $\langle X \rangle^\ell \in \mathbb{Z}_L^{d_2 \times d_3}$ as inputs, and outputs a matrix $\langle Z \rangle^\ell \in \mathbb{Z}_L^{d_1 \times d_3}$ such that $Z = W \boxtimes_\ell X$. We use the OT-based MUX protocol from [99] to implement $\mathcal{F}_{\mathsf{BitMatMul}}^{\ell,d_1,d_2,d_3}$, and also leverage the batching technique here to reduce the number of OTs. The communication required by this protocol is $2 d_1 d_2 (\lambda + 2\ell d_3)$ bits.

Our complete protocol for $\mathcal{F}_{\mathsf{UMatMul}}^{m,n,d_1,d_2,d_3}$ is presented in Algorithm 5 for the $m \leqslant n$ case. The total communication cost of this protocol is $d_1 d_2 d_3 ((2m+4)(n+e) + m^2 + 5m) + d_1 d_2 (\lambda(3m+6) + 14m + e - 6) + d_2 d_3 (\lambda(n+2) + 14n)$ bits. In the protocol, we extend Y because it has elements of larger bitwidth, and this strategy leads to better overall communication in most cases. The other case of $m > n$ is similar and we extend the entries of matrix X by e bits.

E. Digit Decomposition

We consider the functionality $\mathcal{F}_{\mathsf{DigDec}}^{\ell,\{d_i\}_{i \in [c]}}$ that decomposes an ℓ-bit number into c sub-strings or digits of lengths $\{d_i\}$ such that $\sum_{i \in [c]} d_i = \ell$. More formally, $\mathcal{F}_{\mathsf{DigDec}}^{\ell,\{d_i\}_{i \in [c]}}$ takes $\langle x \rangle^\ell$ as input and outputs $\langle z_{c-1} \rangle^{d_{c-1}}, \ldots, \langle z_0 \rangle^{d_0}$ such that $x = z_{c-1} || \ldots || z_0$. We use this functionality in extracting digits to be used as input to lookup tables for approximations for exponential, initial approximation of reciprocal in sigmoid/tanh and reciprocal square root.

For ease of exposition we first consider a simplified functionality $\mathcal{F}_{\mathsf{DigDec}}^{\ell,d}$ with $d \mid \ell$ that outputs $c = \ell/d$ digits of equal length d and present our protocol for this functionality in Algorithm 6. Idea is as follows: To compute the shares of z_i, it suffices to compute the carry of lower bits into this digit when reconstructing shares of x. That is, consider a parsing of ℓ-bit string $\langle x \rangle_b^\ell$ as $y_{b,c-1} || \ldots || y_{b,0}$ such that $y_{b,i} \in \{0,1\}^d$ for all $i \in [c]$ for $b \in \{0,1\}$. Also, set $Y_{b,i} = y_{b,i} || \ldots || y_{b,0}$ for all $i \in [c]$, $b \in \{0,1\}$. Now, observe that $z_i = y_{0,i} + y_{1,i} + \mathsf{carry}_i \bmod 2^d$, where $\mathsf{carry}_i = Y_{0,i-1} + Y_{1,i-1} \geqslant 2^{id}$. Alternatively, $\mathsf{carry}_i = \mathsf{wrap}(Y_{0,i-1}, Y_{1,i-1}, 2^{id})$. In our protocol, we compute this carry_i using Lemma 1 iteratively (similar to our protocol for $\mathcal{F}_{\mathsf{LRS}}^{\ell,s}$) and the variable u_i corresponds to carry_i. The communication complexity of our protocol for the simplified setting is $(c-1)(\lambda(d+2) + 15d + 20)$ bits.

Also, it is easy to see that the above protocol generalizes to the case of unequal size digits, by parsing the initial shares appropriately and doing the same computation. The communication for the generalized case is $\sum_{i \in [c-1]} (\lambda(d_i + 2) + 15 d_i + 20)$ bits. In contrast, doing a digit-decomposition using GC would require $\lambda(6\ell - 2c - 2)$ bits of communication. For example, for $\ell = 32$ and $d = 8$, our protocol has an improvement of $5.5\times$ over GC.

F. Most Significant Non-zero Bit (MSNZB)

For an ℓ-bit integer x, $\mathsf{MSNZB}(x)$ refers to the index of the most significant non-zero-bit. That is, $\mathsf{MSNZB}(x) = k \in [\ell]$, if $x_k = 1$ and $x_j = 0$ for all $j > k$. Alternatively, $\mathsf{MSNZB}(x) = k$ if and only if $2^k \leqslant x < 2^{k+1}$. For the special case of input being 0, $\mathsf{MSNZB}(0) = 0$. Consider the functionality $\mathcal{F}_{\mathsf{MSNZB}}^\ell$ that takes as input $\langle x \rangle^\ell$ and outputs $\{\langle z_i \rangle^B\}_{i \in [\ell]}$ such that

---

Algorithm 5 Unsigned Matrix Multiplication, $\Pi_{\mathsf{UMatMul}}^{m,n,d_1,d_2,d_3}$:

Input: P_0 & P_1 hold $\langle X \rangle^m$ and $\langle Y \rangle^n$, where $X \in \mathbb{Z}_M^{d_1 \times d_2}$, $Y \in \mathbb{Z}_N^{d_2 \times d_3}$ and $m \leqslant n$.

Output: P_0 & P_1 get $\langle Z \rangle^\ell$, where $Z = X \boxtimes_\ell Y$, $\ell = m + n + e$ and $e = \lceil \log d_2 \rceil$.

1: P_0 & P_1 invoke $\mathcal{F}_{\mathsf{ZExt}}^{n,n+e}(\langle Y \rangle^n)$ and learn $\langle Y' \rangle^{n'}$.
2: For $b \in \{0,1\}$, let $X_b = \langle X \rangle_b^m$ and $Y_b' = \langle Y' \rangle_b^{n'}$.
3: P_0 and P_1 invoke the following functionalities.
4: $\quad \mathcal{F}_{\mathsf{MatCrossTerm}}^{m,n',d_1,d_2,d_3}(X_0, Y_1')$ and learn $\langle C \rangle^\ell$.
5: $\quad \mathcal{F}_{\mathsf{MatCrossTerm}}^{n',m,d_3,d_2,d_1}({Y'}_0^T, X_1^T)$ and learn $\langle D \rangle^\ell$.
6: $\quad \mathcal{F}_{\mathsf{Wrap}}^m(X_0, X_1)$ to learn $\langle W_X \rangle^B$.
7: $\quad \mathcal{F}_{\mathsf{Wrap}}^{n'}(Y_0', Y_1')$ to learn $\langle W_{Y'} \rangle^B$.
8: $\quad \mathcal{F}_{\mathsf{BitMatMul}}^{m,d_3,d_2,d_1}(\langle W_{Y'}^T \rangle^B, \langle X^T \rangle^m)$ to learn $\langle G \rangle^m$.
9: $\quad \mathcal{F}_{\mathsf{BitMatMul}}^{n',d_1,d_2,d_3}(\langle W_X \rangle^B, \langle Y' \rangle^{n'})$ to learn $\langle H \rangle^{n'}$.
10: P_b outputs $X_b \boxtimes_\ell Y_b' + \langle C \rangle_b^\ell + \langle D^T \rangle_b^\ell - 2^{n'} *_\ell \langle G^T \rangle_b^m - 2^m *_\ell \langle H \rangle_b^{n'}$ for $b \in \{0,1\}$.

Protocol	Comm. (bits)	Rounds
$\Pi^{m,n}_{\mathsf{ZExt}}$ & $\Pi^{m,n}_{\mathsf{SExt}}$	$\lambda(m+1)+13m+n$	$\log m+2$
$^\star\Pi^{m,n}_{\mathsf{ZExt}}$ & $^\star\Pi^{m,n}_{\mathsf{SExt}}$	$2\lambda-m+n+2$	4
$\Pi^{\ell,s}_{\mathsf{LRS}}$ & $\Pi^{\ell,s}_{\mathsf{ARS}}$	$\lambda(\ell+3)+15\ell+s+20$	$\log \ell+3$
$^\star\Pi^{\ell,s}_{\mathsf{LRS}}$ & $^\star\Pi^{\ell,s}_{\mathsf{ARS}}$	$\lambda(s+3)+\ell+15s+2$	$\log s+2$
$\Pi^{\ell,s}_{\mathsf{TR}}$	$\lambda(s+1)+\ell+13s$	$\log s+2$
$\Pi^{\ell,s}_{\mathsf{DivPow2}}$	$\lambda(\ell+7s/4+4)+16\ell+23s-5$	$\log \ell+4$
$\Pi^{m,n}_{\mathsf{UMult}}$ & $\Pi^{m,n}_{\mathsf{SMult}}$	$\lambda(3\mu+\nu+4)+2\mu\nu+\mu^2+17\mu+16\nu$	$\log \nu+2$
$^\star\Pi^{m,n}_{\mathsf{UMult}}$ & $^\star\Pi^{m,n}_{\mathsf{SMult}}$	$\lambda(2\mu+6)+2\mu\nu+\mu^2+3\mu+2\nu+4$	4
$\Pi^{\ell,d}_{\mathsf{DigDec}}$	$(\ell/d-1)(\lambda(d+2)+15d+20)$	$\log d+\ell/d+1$
$\Pi^{\ell,d}_{\mathsf{MSNZB}}$	$(\ell/d-1)(\lambda(d+8)+2^d(\iota+1)+15d+2\iota+60)+6\lambda+2^d(\iota+1)+\ell^2+2\iota$	$\log d+2\ell/d+7$

TABLE V: Exact communication and round expressions for our building blocks, assuming that the cost of $\Pi^{\ell}_{\mathsf{Mill}}$ and $\Pi^{\ell}_{\mathsf{Mill\&Eq}}$ is $\lambda\ell+14\ell$ bits. $\mu=\mathsf{min}(m,n), \nu=\mathsf{max}(m,n)$, and $\star$ denotes the variant of the protocol in which the MSBs of the inputs are already known in the clear. In case the MSBs are known in the shared form, the additional cost is just $\lambda+2$ bits per input.

Algorithm 6 Digit Decomposition, $\Pi^{\ell,d}_{\mathsf{DigDec}}$:

Input: P_0 & P_1 hold $\langle x\rangle^\ell$ s.t. $c=\ell/d$.
Output: P_0 & P_1 get $\{\langle z_i\rangle^d\}_{i\in[c]}$ s.t. $x=z_{c-1}||\ldots||z_0$.

1: For $b\in\{0,1\}$, P_b parses $\langle x\rangle^\ell_b$ as an ℓ-bit string $y_{b,c-1}||\ldots||y_{b,0}$ s.t. $y_{b,i}\in\{0,1\}^d$ for all $i\in[c]$.
2: For all $i\in\{0,\ldots,c-2\}$, P_0 & P_1 invoke $\mathcal{F}^d_{\mathsf{Wrap\&All1s}}(y_{b,i},y_{b,1})$ and learn $\langle w_i\rangle^B||\langle e_i\rangle^B$.
3: For $b\in\{0,1\}$, P_b sets $\langle u_0\rangle^B_b=0$ and $\langle z_0\rangle^d_b=y_{b,0}$.
4: **for** $i\in\{1,\ldots,c-1\}$ **do**
5: $\quad P_0$ & P_1 invoke $\mathcal{F}_{\mathsf{AND}}(\langle u_{i-1}\rangle^B,\langle e_{i-1}\rangle^B)$ to learn $\langle v_{i-1}\rangle^B$.
6: $\quad$ For $b\in\{0,1\}$, P_b sets $\langle u_i\rangle^B_b=\langle v_{i-1}\rangle^B_b\oplus\langle w_{i-1}\rangle^B_b$.
7: $\quad P_0$ & P_1 invoke $\mathcal{F}^d_{\mathsf{B2A}}(\langle u_i\rangle^B)$ and learn $\langle u_i\rangle^d$.
8: $\quad$ For $b\in\{0,1\}$, P_b sets $\langle z_i\rangle^d_b=y_{b,i}+\langle u_i\rangle^B_b$.
9: **end for**

$z_i=1$ if $\mathsf{MSNZB}(x)=i$ and 0 otherwise. Our protocol for $\mathcal{F}^\ell_{\mathsf{MSNZB}}$ reduces to MSNZB-like computation on integers on smaller bit-length as follows: For simplicity of exposition, consider $d\in\mathbb{N}$ such that $d\mid\ell$. First, we invoke $\mathcal{F}^{\ell,d}_{\mathsf{DigDec}}$ to decompose ℓ-bit integer x into $c=\ell/d$ integers of d-bits, say $\{y_i\}_{i\in[c]}$. Now, we compute MSNZB on each of these smaller integers y_i by taking into account their position i in x and output an index in $[\ell]$ which corresponds to $\mathsf{MSNZB}(y_i)+i\cdot d$. Note that $\mathsf{MSNZB}(x)=\mathsf{MSNZB}(y_i)+i\cdot d$ if $y_i\neq 0$ and $y_j=0$ for all $j>i$. To realize this logic we also compute whether $y_i=0$ for all $i\in[c]$.

More formally, let $\iota=\log\ell$ and consider the functionality $\mathcal{F}^{d,\ell,i}_{\mathsf{MSNZB\text{-}P}}$ for $i\in[c]$ that takes as input $\langle y\rangle^d$ and outputs $\langle u\rangle^\iota$ such that $2^{u-id}\leqslant y<2^{u-id+1}$. Also, consider $\mathcal{F}^d_{\mathsf{Zeros}}$ functionality that takes as input $\langle y\rangle^d$ and outputs $\langle v\rangle^B$ such that $v=\mathbf{1}\{y=0\}$. First, our protocol invokes $\mathcal{F}^{d,\ell,i}_{\mathsf{MSNZB\text{-}P}}$ on each of $\langle y_i\rangle^d$ (obtained from $\mathcal{F}^{\ell,d}_{\mathsf{DigDec}}(\langle x\rangle^\ell)$) to learn $\langle u_i\rangle^\iota$. Next, we invoke $\mathcal{F}^d_{\mathsf{Zeros}}(y_i)$ to learn $\langle v_i\rangle^B$. Now, for all $i\in[c]$, we compute $z'_i=u_i\cdot(1\oplus v_i)\cdot\prod_{j>i}v_j$. Note that $z'_i=u_i$ if $y_i\neq 0$ and $y_j=0$ for all $j>i$ and 0 otherwise. Moreover, at most one z'_i is non-zero. Hence, we compute $\mathsf{MSNZB}(x)=\tilde{z}=\sum_i z'_i$. Finally, to output the one-hot encoding described above, we invoke the functionality $\mathcal{F}^\ell_{\mathsf{One\text{-}Hot}}$ that takes as input $\langle\tilde{z}\rangle^\iota$ and outputs $\{\langle z_i\rangle^B\}_{i\in[\ell]}$ such that $z_i=1$ for $i=\tilde{z}$ and 0 otherwise. We present our protocol for $\mathcal{F}^\ell_{\mathsf{MSNZB}}$ in Algorithm 7, for the special case of $d\mid\ell$; it is easy to see that the general case works in a similar manner. Our protocol makes 1 call to $\mathcal{F}^{\ell,d}_{\mathsf{DigDec}}$, c calls each to $\mathcal{F}^{d,\ell,i}_{\mathsf{MSNZB\text{-}P}}$, $\mathcal{F}^d_{\mathsf{Zeros}}$ (with i going from 0 to $c-1$) and $\mathcal{F}^\iota_{\mathsf{MUX}}$, $2c-2$ calls to $\mathcal{F}_{\mathsf{AND}}$ and 1 call to $\mathcal{F}^\ell_{\mathsf{One\text{-}Hot}}$.

We implement both $\mathcal{F}^{d,\ell,i}_{\mathsf{MSNZB\text{-}P}}$ and $\mathcal{F}^d_{\mathsf{Zeros}}$ using LUTs with d-bit inputs. Moreover, since these are invoked on same input, we combine them into a single LUT with entries $(u_i||v_i)$. Finally, we implement $\mathcal{F}^\ell_{\mathsf{One\text{-}Hot}}$ using an LUT with ι-bit input and ℓ-bit entries. The exact expression for communication for $d\mid\ell$ is given in Table V. The expression for the general case can be computed similarly using expression in digit decomposition. Based on empirical findings, we use $d=8$ in our implementation.

Algorithm 7 Most Significant Non-Zero Bit, $\Pi^{\ell,d}_{\mathsf{MSNZB}}$:

Input: For $b\in\{0,1\}$, P_b holds $\langle x\rangle^\ell_b$, $c=\ell/d, \iota=\log\ell$.
Output: For $b\in\{0,1\}$, P_b learns $\{\langle z_i\rangle^B_b\}_{i\in[\ell]}$ s.t. $z_i=1$ if $2^i\leqslant x<2^{i+1}$ and 0 otherwise.

1: P_0 & P_1 invoke $\mathcal{F}^{\ell,d}_{\mathsf{DigDec}}(\langle x\rangle^\ell)$ and learn $\{\langle y_i\rangle^d\}_{i\in[c]}$.
2: **for** $i\in\{0,\ldots,c-1\}$ **do**
3: $\quad P_0$ & P_1 invoke $\mathcal{F}^{d,\ell,i}_{\mathsf{MSNZB\text{-}P}}(\langle y_i\rangle^d)$ and learn $\langle u_i\rangle^\iota$.
4: $\quad P_0$ & P_1 invoke $\mathcal{F}^d_{\mathsf{Zeros}}(\langle y_i\rangle^d)$ and learn $\langle v_i\rangle^B$.
5: $\quad$ For $b\in\{0,1\}$, P_b sets $\langle v'_i\rangle^B_b=(b\oplus\langle v_i\rangle^B_b)$.
6: **end for**
7: P_0 & P_1 invoke $\mathcal{F}^\iota_{\mathsf{MUX}}(\langle v'_{c-1}\rangle^B,\langle u_{c-1}\rangle^\iota)$ and learn $\langle z'_{c-1}\rangle^\iota$.
8: For $b\in\{0,1\}$, P_b sets $\langle w_{c-1}\rangle^B_b=b$.
9: **for** $i\in\{c-2,\ldots,0\}$ **do**
10: $\quad P_0$ & P_1 invoke $\mathcal{F}_{\mathsf{AND}}(\langle w_{i+1}\rangle^B,\langle v_{i+1}\rangle^B)$ and learn $\langle w_i\rangle^B$.
11: $\quad P_0$ & P_1 invoke $\mathcal{F}_{\mathsf{AND}}(\langle w_i\rangle^B,\langle v'_i\rangle^B)$ and learn $\langle w'_i\rangle^B$.
12: $\quad P_0$ & P_1 invoke $\mathcal{F}^\iota_{\mathsf{MUX}}(\langle w'_i\rangle^B,\langle u_i\rangle^\iota)$ and learn $\langle z'_i\rangle^\iota$.
13: **end for**
14: For $b\in\{0,1\}$, P_b sets $\langle\tilde{z}\rangle^\iota_b=\sum_{i=0}^{c-1}\langle z'_i\rangle^\iota_b$.
15: P_0 & P_1 invoke $\mathcal{F}^\ell_{\mathsf{One\text{-}Hot}}(\langle\tilde{z}\rangle^\iota)$ and learn $\{\langle z_i\rangle^B\}_{i\in[\ell]}$.