Kanav Gupta*, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta

LLAMA: A Low Latency Math Library for Secure Inference

Abstract: Secure machine learning (ML) inference can provide meaningful privacy guarantees to both the client (holding sensitive input) and the server (holding sensitive weights of the ML model) while realizing inference-as-a-service. Although many specialized protocols exist for this task, including those in the preprocessing model (where a majority of the overheads are moved to an input independent offline phase), they all still suffer from large online complexity. Specifically, the protocol phase that executes once the parties know their inputs, has high communication, round complexity, and latency. Function Secret Sharing (FSS) based techniques offer an attractive solution to this in the trusted dealer model (where a dealer provides input independent correlated randomness to both parties), and 2PC protocols obtained based on these techniques have a very lightweight online phase.

Unfortunately, current FSS-based 2PC works (AriaNN, PoPETS 2022; Boyle *et al.* Eurocrypt 2021; Boyle *et al.* TCC 2019) fall short of providing a complete solution to secure inference. First, they lack support for math functions (e.g., sigmoid, and reciprocal square root) and hence, are insufficient for a large class of inference algorithms (e.g. recurrent neural networks). Second, they restrict all values in the computation to be of the same bitwidth and this prevents them from benefitting from efficient float-to-fixed converters such as Tensorflow Lite that crucially use low bitwidth representations and mixed bitwidth arithmetic.

In this work, we present LLAMA – an end-to-end, FSS based, secure inference library supporting precise low bitwidth computations (required by converters) as well as *provably precise* math functions; thus, overcoming all the drawbacks listed above. We perform an extensive evaluation of LLAMA and show that when compared with non-FSS based libraries supporting mixed bitwidth arithmetic and math functions (SIRNN, IEEE S&P 2021), it has at least an order of magnitude lower communication, rounds, and runtimes. We integrate LLAMA with the EzPC framework (IEEE EuroS&P 2019) and demonstrate its robustness by evaluating it on large benchmarks (such as ResNet-50 on the ImageNet dataset) as well as on benchmarks considered in AriaNN – here too LLAMA outperforms prior work.

Keywords: Function Secret Sharing, Secure Inference, Secure Two-Party Computation

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

1 Introduction

One of the most important security problems in the area of machine learning is that of *secure inference*, wherein, a model owner S, holding a public model M and its associated private weights w, offers inference-as-a-service to a client C, with private input data point x, with the security guarantee that S learns nothing about x, while C only learns $M(w, x)$, i.e. the output of the model on the input data, and nothing else. Being a special case of the generic problem of secure multi-party computation (MPC or 2PC in the 2-party case) [15, 41], this problem has received widespread attention – both within the cryptography community [19, 20, 28, 29, 31] as well as in several application domains such as healthcare [21, 36].

On the technical side, perhaps the biggest bottleneck towards the widespread adoption of cryptographically secure inference protocols, is their large communication and interaction cost. As an example, even if one were to execute secure inference on a simple 7-layer convolutional neural network (CNN) [27] on the CIFAR-10 dataset using a state-of-the-art system [31], one inference would cost approximately 340 MB of communication, 345 rounds of interaction, and would take about 10 seconds to execute on commodity hardware with good bandwidth. In light of this, works such as DELPHI [28] have crucially focused on reducing the online cost of such secure protocols – that is, where a bulk of the overheads can be moved to an input independent pre-processing phase. However, even here, the situation is far from satisfactory and the above benchmark would still require 196 MB of online communication and 2.7 seconds of online execution time. Further, in-

***Corresponding Author: Kanav Gupta:** Microsoft Research. E-mail: t-kanavgupta@microsoft.com.
Deepak Kumaraswamy: Microsoft Research. E-mail: t-deepakk@microsoft.com.
Nishanth Chandran: Microsoft Research. E-mail: nichandr@microsoft.com.
Divya Gupta: Microsoft Research. E-mail: divya.gupta@microsoft.com.

ference algorithms that contain mathematical functions such as sigmoid and tanh are more expensive to compute securely; e.g. running a recurrent neural network (RNN) [26] on the standard Google-30 dataset [40] to identify commands, directions, and digits from speech costs 415 MB, ≈ 60,000 rounds, and ≈ 37 seconds [30].

1.1 Function Secret Sharing based 2PC

A recently emerging trend to build secure computation protocols with low online cost is through the technique of function secret sharing (FSS) [5, 7]. In this paradigm, a trusted dealer provides input independent correlated randomness to the two parties executing the secure computation protocol. Here, round and communication optimal 2PC protocols are known for many functions such as addition, multiplication, comparison, and so on; thus, leading to 2PC protocols with low online cost for various algorithms [3, 7, 34]. This paradigm, which leads to dramatic improvements in online cost, is practically well motivated too [2, 4, 10, 21, 22] – typically the vendor providing or building the 2PC solution for the client and server in our above example is trusted to provide correct 2PC code and hence providing correlated randomness is not an additional burden.

While FSS-based 2PC protocols for secure computation [7], fixed-point arithmetic [3], and secure inference [34] are known, these works suffer from the following drawbacks.

Lack of math functions support. First, they lack support for math functions (e.g. sigmoid, tanh, reciprocal square root) and thus are incapable of handling benchmarks such as the RNN described above[1]. Recent work [30] provides a secure computation library for precisely computing various math functions by designing approximate functionalities that simultaneously have low error and are also efficient to compute securely in 2PC. However, unfortunately, the protocols used to compute these functionalities require high online communication and rounds (due to the highly sequential nature of the functionalities themselves) and hence, as we note later, the online cost of computing them is still prohibitive. Furthermore, given this, even the functionalities themselves are not the optimal design choice to be computed securely using FSS techniques.

Lack of support for mixed bitwidth arithmetic. Second, most existing MPC frameworks support only uniform bitwidth arithmetic – i.e., a single bitwidth and scale are used uniformly for all fixed-point values and all secure computation must execute within these limits. On the other hand, state-of-the-art converters such as Tensorflow Lite [16] and Shiftry [24], that quantize floating point models to corresponding fixed point ones, for efficiency, use low bitwidth representations (e.g. 8 or 16). To obtain precise outputs of operators such as simple matrix multiplication with low bitwidth inputs and outputs, these libraries rely on using higher bitwidths to compute intermediate results such as element-wise multiplications and accummulations in a dot product to avoid overflows. Later, these intermediate results are quantized to the right output format. However, without the support for such precise computation on low bitwidths (which implicitly require mixed bitwidth arithmetic), systems such as [9] were forced to perform secure computation over large bitwidths, resulting in high performance overhead. The work of [30] was the first to provide 2PC protocols for such precise low bitwidth arithmetic, but these protocols have high online interaction and communication.

Erroneous ReLU and truncations. Finally, the only prior FSS-based secure inference library AriaNN [34], uses protocols for ReLU activations[2] and truncations of fixed-point values (required to maintain scale) that are only probabilistically correct. This has two drawbacks – first, since this error probability is inversely proportional to the bitwidth used, the protocols are forced to use larger bitwidths to ensure that this error probability does not affect the accuracy of the resulting inference algorithm; second, as the number of instances of ReLU/truncation in the algorithm increases, the probability of the overall computation being incorrect increases and hence, as noted in multiple works [9, 19, 25], on large benchmarks, such probabilistic computations would almost certainly provide incorrect outputs.

1.2 Our Contributions

In this work, we address the above drawbacks and present LLAMA – an end-to-end semi-honest secure

1 Recent work of [37], which focuses on application of FSS to recommendation systems, considered only reciprocal square root. We provide a comparison with this work in Section 1.3.

2 The ReLU activation is defined as $\mathsf{ReLU}(x) = \mathsf{max}(x, 0)$.

2PC inference system (in the trusted dealer model) that is based on FSS techniques. LLAMA has significantly lower online complexity compared to previous works on 2-party secure inference [28, 30, 31] and is much more expressive than AriaNN [34], the only prior secure inference system based on FSS. We discuss our technical contributions and features of LLAMA below.

Precise Low Bitwidth Computation. First, LLAMA supports precise secure computation over low bitwidth values. As discussed, this requires computing intermediary results over higher bitwidths and then appropriately reducing these results to lower bitwidths. Supporting this requires providing FSS-based protocols for two crucial functionalities – Zero/Signed-Extension (to increase bitwidth) as well as Truncate-Reduce (to decrease both bitwidth and scale). We design a single-round FSS protocol, also known as an FSS gate, for these operations that act as building blocks in our other protocols (Section 3). Next, we build on these and design protocols for element-wise multiplications and matrix multiplications (and convolutions) that implement the arithmetic logic used in fixed-point implementations of converters [16, 24] (Section 4).

Low Bitwidth Splines and Math Functions. Second, we provide "FSS-friendly" low bitwidth approximations to math functions (such as sigmoid, tanh and reciprocal square root) that are *provably precise*. We use ULPs[3] as the measure of preciseness of math implementations that is also used by standard math libraries and ensure that our implementations have at most 4 ULPs error (similar to math libraries and 2PC work SIRNN [30]). As already mentioned, approximate functionalities provided in SIRNN are highly sequential and would lead to large number of rounds in the online phase even when implemented with FSS-based techniques. Hence, we deviate significantly from SIRNN in our design choice and instead use low bitwidth piecewise polynomials or *splines* to approximate our math functions[4]. However, standard tools for finding splines result in floating-point splines. We convert these to fixed-point splines keeping FSS costs in mind. Next, once we have such a spline, as discussed in Section 5, we need to evaluate it efficiently using FSS techniques. Here, we build upon the work of [3] that provided an FSS protocol for uniform bitwidth spline evaluation and extend it to protocols for low bitwidth splines. Further, LLAMA uses two novel optimizations over [3] that significantly reduce keysize as well as PRG invocations during the online phase of that protocol. As an example, for the spline approximating sigmoid, our techniques reduce keysize by 4× and PRG invocations by 40×.

ReLU, Truncation, Pooling. Third, unlike [34], LLAMA uses correct protocols for comparison and ReLU from [3]. We build on these to provide correct protocols for average pool, maxpool, and argmax (Appendix C). Next, unlike [34], in LLAMA all types of truncations (bitwidth preserving or bitwidth reducing) and comparisons are faithful, resulting in correct computation. With this, our FSS implementations are bitwise equivalent to the corresponding cleartext fixed-point implementations. This enables us to execute large benchmarks with no fear of incorrect outputs (even when using very small bitwidths in some cases) .

E2E Inference System. Fourth, we integrate LLAMA as a cryptographic backend to the EzPC framework [8]. Together, all of the above, enable us to execute various benchmarks securely – large CNNs (such as ResNet-50 on ImageNet dataset) using the CrypTFlow toolchain [25] as well as RNNs (e.g. [26] on the Google-30 dataset) using the SIRNN toolchain [30]. For almost all our benchmarks, we obtain at least an order of magnitude reduction in online communication, rounds and runtimes (Section 6), thus obtaining, a low latency framework for secure inference.

Let us revisit the two examples presented earlier in the introduction. Running LLAMA on the RNN [26] over the Google-30 dataset costs only roughly 8.6 MB of online communication, 2600 online rounds, and 1.9 seconds, resulting in roughly 48×, 22×, and 19× improvement in communication, rounds and performance over SIRNN [30]. Similarly, executing the CNN model [27] on the CIFAR-10 dataset costs 8.25 MB of online communication, and 0.5 seconds resulting in approximately 24× and 5× improvements in online communication and times over DELPHI [28]. We now proceed to introduce all background technical information in the next section.

3 Informally, ULP (unit of least precision) is the number of representable values between our result and output over reals. It is widely accepted as the standard for measuring accuracy in numeric calculations [14]. See Section 5 for more details.

4 We note that although prior works such as SecureML [29] also used a spline to approximate sigmoid, the spline used had only 3 pieces and leads to very high ULP error and hence, a high degradation in classification accuracy as shown in [30].

1.3 Other Related Works

The work on FSS-based 2PC protocols for fixed-point arithmetic by Boyle *et al.* [3] provides FSS gates for various building blocks such as ReLU, arithmetic/logical right shift, and splines. However, [3] lacks support for FSS-gates of signed-extension, truncate-reduce and hence, does not support mixed-bitwidth operations and precise math functions over small bitwidths.

A recent work by Vadapalli *et al.* [37] also use spline to compute reciprocal square root and realize it using DPF (Distributed Point Function) [6]. Our work and [37] present a trade-off between online computation and key size. The online compute in LLAMA grows proportional to the number of intervals of the spline. On the other hand, the online compute of [37] grows exponentially with the input bitwidth. (For reciprocal square root with 16-bit inputs, in the online phase, LLAMA makes 1448 AES evaluations, and [37] makes 131072 AES evaluations.) However, the key size in LLAMA is higher than that in [37]. (For reciprocal square root with 16-bit inputs, key size in LLAMA is nearly 5KB, compared to around 0.3KB in [37].) Since the primary benefit of FSS based techniques is to reduce online complexity, LLAMA would perform better than [37].

2 Preliminaries

Notation. Let λ denote the computational security parameter. We use uppercase L, M, N to denote the values $2^\ell, 2^m$ and 2^n respectively. $\mathbf{1}\{b\}$ denotes the indicator function that is 1 when b is true and 0 otherwise. We use the natural one to one map between $\{0,1\}^\ell$ and $\mathbb{Z}_L$.

We consider computations over finite bit unsigned and signed integers, denoted by $\mathbb{U}_N$ and $\mathbb{S}_N$, respectively, for a given bitwidth of n-bits. We note that $\mathbb{U}_N = \{0, \ldots, N-1\}$ is isomorphic to $\mathbb{Z}_N$. Signed integers $\mathbb{S}_N$ range from $-N/2$ till $N/2-1$ and can be encoded into $\mathbb{Z}_N$ or $\mathbb{U}_N$ using 2's complement representation. In this encoding, the MSB of the bit-representation of $x \in \mathbb{S}_N$ is 0 if $x \geqslant 0$ and 1 otherwise. We use the functions $\mathsf{uint}_n : \mathbb{U}_N \to \mathbb{Z}$ and $\mathsf{sint}_n : \mathbb{S}_N \to \mathbb{Z}$ to represent the conversion of a number to its unsigned and signed number in $\mathbb{Z}$ respectively. In 2's complement notation, $\mathsf{sint}_n(x) = \mathsf{uint}_n(x) - \mathsf{MSB}(x) * N$ where $\mathsf{MSB}(x) = \mathbf{1}\{x \geqslant 2^{n-1}\}$. We drop the subscript whenever the bitwidth can be inferred from the context. We also use the symbol $*_\ell : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}_L$ to denote the binary operation $x *_\ell y = x \cdot y \mod L$ where $x, y \in \mathbb{Z}$. For $x \in \mathbb{S}_N$, we use the notation of $x[i]$ to represent the i-th bit from the LSB in the 2's complement representation of x such that LSB is $x[0]$ and MSB is $x[n-1]$. We also use $x_{[i,j)} \in \mathbb{Z}_{2^{j-i}}$ (where $j > i$) to denote the number formed by the bitstring $x[j-1], x[j-2] \ldots x[i]$.

Fixed-point representation. Real numbers are encoded into $\mathbb{Z}_N$ using fixed-point notation. The fixed-point numbers are parameterized by two values, a bitwidth n and a scale s. The first $n-s$ bits and last s bits correspond to the integer part and the fractional part respectively. We have $y = \mathsf{fix}_{n,s}(x) = \lfloor x \cdot 2^s \rfloor \mod N$. To convert a fixed-point integer y to its real counterpart x, we have $x = \mathsf{urt}_{n,s}(y) = \mathsf{uint}(y)/2^s$ if $y \in \mathbb{U}_N$ and $x = \mathsf{srt}_{n,s}(y) = \mathsf{sint}(y)/2^s$ if $y \in \mathbb{S}_N$.

2.1 Function Secret Sharing (FSS)

An FSS scheme [5, 6] is a pair of algorithms, namely Gen and Eval. Gen splits a secret function $f : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$ into a pair of functions f_0 and f_1. For the party identifier $\sigma \in \{0,1\}$, Eval evaluates the function f_σ on a given input $x \in \mathbb{G}^{\mathsf{in}}$. While correctness of an FSS scheme requires that $f_0(x) + f_1(x) = f(x)$ for all $x \in \mathbb{G}^{\mathsf{in}}$, the security requires that each f_σ hides f.

Definition 1 (FSS: Syntax [5, 6])**.** *A (2-party)* function secret sharing scheme *is a pair of algorithms* $(\mathsf{Gen}, \mathsf{Eval})$ *such that:*

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

Definition 2 (FSS: Correctness and Security [5, 6])**.** *Let* $\mathcal{F} = \{f\}$ *be a function family and* Leak *be a function specifying the allowable leakage about* $\hat{f}$*. When* Leak *is omitted, it is understood to output only* $\mathbb{G}^{\mathsf{in}}$ *and* $\mathbb{G}^{\mathsf{out}}$*. We say that* $(\mathsf{Gen}, \mathsf{Eval})$ *as in Definition 1 is an* FSS scheme for $\mathcal{F}$ *(with respect to leakage* Leak*) if it satisfies the following requirements.*

- **Correctness:** *For all* $\hat{f} \in P_{\mathcal{F}}$ *describing* $f : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}$*, and every* $x \in \mathbb{G}^{\mathsf{in}}$*, if* $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f})$ *then* $\Pr\left[\mathsf{Eval}(0, k_0, x) + \mathsf{Eval}(1, k_1, x) = f(x)\right] = 1$*.*
- **Security:** *For each* $\sigma \in \{0,1\}$ *there is a PPT algorithm* Sim_σ *(simulator), such that for every se-*

quence $(\hat{f}_\lambda)_{\lambda\in\mathbb{N}}$ *of polynomial-size function descriptions from* $\mathcal{F}$ *and polynomial-size input sequence* x_λ *for* f_λ*, the outputs of the following experiments* Real *and* Ideal *are computationally indistinguishable:*

- Real_λ*:* $(k_0, k_1) \leftarrow \mathsf{Gen}(1^\lambda, \hat{f}_\lambda)$*; Output* k_σ*.*
- Ideal_λ*: Output* $\mathsf{Sim}_\sigma(1^\lambda, \mathsf{Leak}(\hat{f}_\lambda))$*.*

2.2 2PC with preprocessing via FSS

Threat Model. We consider 2PC with a trusted dealer secure against a static PPT adversary that corrupts one of the parties. That is, the adversary is computationally bounded, corrupts one of the parties at the beginning of the protocol, and follows the protocol specification. The dealer gives out the input independent correlated randomness to both the parties in the offline phase. Given the correlated randomness, the parties engage in a 2PC protocol in the online phase. We consider standard simulation paradigm for semi-honest security.

Boyle *et al.* [7] construct 2PC protocols (in a trusted dealer model, where the dealer provides correlated randomness to the 2 parties[5]) via FSS. The high level idea is as follows: For each wire w_i in the circuit to be computed, the dealer picks a mask value r_i uniformly at random. Denote the cleartext value at w_i by x_i. The protocol maintains the invariant that the 2 parties hold masked values to input wire of a gate, and run FSS evaluation protocol to learn masked value of the output wire with one round of simultaneous message exchange. In more detail, to compute a gate g_{ij} with input and output wires w_i and w_j, parties start with x_i+r_i and end up with $x_j + r_j$ after one round of interaction. Moreoever, the size of the message exchanged is the same as the bitwidth of x_j. To enable this, the dealer gives out FSS keys for the *offset function* $g_{ij}^{[\mathsf{r}^{in},\mathsf{r}^{out}]}(x) = g_{ij}(x-r_i)+r_j$ in the pre-processing phase. In the online phase, the parties compute their share of the function on the masked input $x_i + r_i$ to obtain the secret shares of the value $x_j + r_j$, which they reconstruct to obtain the masked output value. For the input wires, dealer simply sends the masks of the wire to its respective owner, who on receiving the mask, adds it to the input and sends it to the other party. For the output wire, the dealer sends the mask to both of the parties. For more details on the construction of 2PC protocols using FSS, we refer the reader to [7]. Now we formally define FSS gates:

Definition 3 (FSS Gates [3])**.** *Let* $\mathcal{G} = \{g : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}}\}$ *be a computation gate (parameterized by input and output groups* $\mathbb{G}^{\mathsf{in}}, \mathbb{G}^{\mathsf{out}}$*). The family of offset functions* $\hat{\mathcal{G}}$ *of* $\mathcal{G}$ *is given by*

$$\hat{\mathcal{G}} := \left\{ g^{[\mathsf{r}^{in},\mathsf{r}^{out}]} : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}} \;\middle|\; \begin{array}{c} g : \mathbb{G}^{\mathsf{in}} \to \mathbb{G}^{\mathsf{out}} \in \mathcal{G}, \\ \mathsf{r}^{in} \in \mathbb{G}^{\mathsf{in}}, \mathsf{r}^{out} \in \mathbb{G}^{\mathsf{out}} \end{array} \right\}$$

$$\text{where } g^{[\mathsf{r}^{in},\mathsf{r}^{out}]}(x) := g(x - \mathsf{r}^{in}) + \mathsf{r}^{out},$$

and $g^{[\mathsf{r}^{in},\mathsf{r}^{out}]}$ *contains an explicit description of* $\mathsf{r}^{in}, \mathsf{r}^{out}$*. Finally, we use the term FSS gate for* $\mathcal{G}$ *to denote an FSS scheme for the corresponding offset family* $\hat{\mathcal{G}}$*.*

2.3 Prior FSS schemes and FSS gates

In this section, we discuss the FSS schemes and FSS gates constucted in prior works [3, 6] that serve as building blocks for us. A distributed comparison function is an FSS scheme for special intervals as defined below.

Definition 4 (DCF [3, 6])**.** *A* special interval *function* $f^{<}_{\alpha,\beta}$*, also referred to as a* comparison *function, outputs* β *if* $x < \alpha$ *and 0 otherwise. We refer to an FSS schemes for comparison functions as* distributed comparison function *(DCF). Analogously, function* $f^{\leqslant}_{\alpha,\beta}$ *outputs* β *if* $x \leqslant \alpha$ *and* 0 *otherwise. In all of these cases, we allow the default leakage* $\mathsf{Leak}(\hat{f}) = (\mathbb{G}^{\mathsf{in}}, \mathbb{G}^{\mathsf{out}})$*.*

For $\alpha \in \{0,1\}^n$ and $\beta \in \mathbb{G}$, we use $\mathsf{Gen}^{<}_n(1^\lambda, \alpha, \beta, \mathbb{G})$ and $\mathsf{Eval}^{<}_n(b, k_b, x)$ to denote the keygen and evaluation algorithms for DCF.

Theorem 1 (Concrete cost of DCF [3])**.** *Let* λ *be the security parameter,* $\mathbb{G}$ *be an Abelian group, and* $\ell = \lceil \log |\mathbb{G}| \rceil$*. Given a PRG* $G : \{0,1\}^\lambda \to \{0,1\}^{4\lambda+2}$*, there exists a DCF for* $f^{<}_{\alpha,\beta} : \{0,1\}^n \to \mathbb{G}$ *with key size* $n(\lambda+\ell+2)+\lambda+\ell$ *bits. For* $\ell' = \lceil \frac{\ell}{4\lambda+2} \rceil$*, the key generation algorithm* $\mathsf{Gen}^{<}_n$ *invokes* G *at most* $2n(1+2\ell')+2\ell'$ *times and the evaluation algorithm* $\mathsf{Eval}^{<}_n$ *invokes* G *at most* $n(1+\ell')+\ell'$ *times. In the special case that* $|\mathbb{G}| = 2^c$ *for* $c \leqslant \lambda$ *the number of PRG invocations in* $\mathsf{Gen}^{<}_n$ *is* $2n$ *and the number of PRG invocations in* $\mathsf{Eval}^{<}_n$ *is* n*.*

Dual Distributed Comparison Function (DDCF) is a generalization of DCF also introduced in [3]. It is defined as a class of comparison functions $f^{\mathsf{DDCF}}_{\alpha,\beta_1,\beta_2} : \{0,1\}^n \to \mathbb{G}$ which returns β_1 if the input value is less than α and β_2 otherwise. DDCF can be described in

5 [3] discuss how the pre-processing FSS keys can be generated using standard 2PC, but in this work we focus on 2PC with a trusted dealer setting.

terms of DCF by $f^{\mathsf{DDCF}}_{\alpha,\beta_1,\beta_2}(x) = \beta_2 + f^{<}_{\alpha,\beta_1-\beta_2}(x)$. We use $\mathsf{Gen}^{\mathsf{DDCF}}_n(1^\lambda, \alpha, \beta_1, \beta_2, \mathbb{G})$ and $\mathsf{Eval}^{\mathsf{DDCF}}_n(b, k_b, x)$ to denote FSS algorithms for DDCF.

We abuse notation and use $\mathsf{DCF}_{n,\mathbb{G}}$ and $\mathsf{DDCF}_{n,\mathbb{G}}$ to denote cost (either keysize or evaluation cost) of FSS schemes for DCF and DDCF respectively.

FSS gates. [3, 7] construct FSS gates for uniform bitwidth matrix multiplication, interval containment, signed comparison and splines. We summarize their costs in Table 1.

3 Bitwidth-Changing Gates

Mixed-bitwidth arithmetic (multiplication, spline evaluations and so on) fundamentally relies on two building blocks – Extension, that increases the bitwidth of an input from m to n; and Truncate-Reduce, that truncates and reduces the bitwidth of the input from n to $n-s$. In this section, we present FSS gates for Extension (in Section 3.1) and Truncate-Reduce (in Section 3.2).

3.1 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 $x \in \mathbb{U}_M$ (resp. $x \in \mathbb{S}_M$), Zero-Extension (resp. Signed-Extension) to n-bits ($n > m$) is defined by $y = \mathsf{ZExt}(x, m, n) \in \mathbb{U}_N$ (resp. $y = \mathsf{SExt}(x, m, n) \in \mathbb{S}_N$), such that $\mathsf{uint}_n(y) = \mathsf{uint}_m(x)$ (resp. $\mathsf{sint}_n(y) = \mathsf{sint}_m(x)$) holds. In the discussion that follows, we only consider the case of Signed-Extension and note that the protocol for Zero-Extension can be derived similarly.

The Signed-Extension gate $\mathcal{G}_{\mathsf{SExt}}$ is the family of functions $g_{\mathsf{SExt},m,n} : \mathbb{S}_M \to \mathbb{S}_N$ parameterized by input group $\mathbb{G}^{\mathsf{in}} = \mathbb{S}_M$ and output group $\mathbb{G}^{\mathsf{out}} = \mathbb{S}_N$, and defined as $g_{\mathsf{SExt},m,n}(z) := \mathsf{sint}_m(z) \mod N$. For this gate (and further gates described in the coming sections), we use the following equation for sint_m from [30]:

$$\mathsf{sint}_m(z) = z' - 2^{m-1}, \text{ for } z' = z + 2^{m-1} \mod M \quad (1)$$

We denote the corresponding offset gate class by $\hat{\mathcal{G}}_{\mathsf{SExt}}$ and the offset functions by:

$$\begin{aligned}\hat{g}^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{\mathsf{SExt},m,n}(x) &= g_{\mathsf{SExt},m,n}(x - \mathsf{r}^{\mathsf{in}}) + \mathsf{r}^{\mathsf{out}} \\ &= \mathsf{sint}_m((x - \mathsf{r}^{\mathsf{in}}) \mod M) + \mathsf{r}^{\mathsf{out}} \\ &= (x' - \mathsf{r}^{\mathsf{in}}) \mod M - 2^{m-1} + \mathsf{r}^{\mathsf{out}}\end{aligned}$$

where $x' = x + 2^{m-1} \mod M$. Observe that for $a, b \in \mathbb{U}_M$, the following equations holds for arithmetic in $\mathbb{Z}$:

$$(a-b) \mod M = a - b + M \cdot \mathbf{1}\{a < b\} \quad (2)$$

So, on using the above equation, we get:

$$\hat{g}^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{\mathsf{SExt},m,n}(x) = x' - \mathsf{r}^{\mathsf{in}} + M \cdot \mathbf{1}\{x' < \mathsf{r}^{\mathsf{in}}\} - 2^{m-1} + \mathsf{r}^{\mathsf{out}} \quad (3)$$

We present our construction of the FSS gate for Signed-Extension in Figure 1 and provide the summary of cost along with proof of security in Theorem 2. The proofs for subsequent FSS protocols in our paper can be derived in a similar manner, and we omit them.

Signed-Extension Gate $(\mathsf{Gen}^{\mathsf{SExt}}_{m,n}, \mathsf{Eval}^{\mathsf{SExt}}_{m,n})$

$\mathsf{Gen}^{\mathsf{SExt}}_{m,n}(1^\lambda, \mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:

1: Sample random $r_0, r_1 \leftarrow \mathbb{S}_N$ s.t. $r_0 + r_1 = \mathsf{sint}_n(\mathsf{r}^{\mathsf{out}}) - \mathsf{sint}_m(\mathsf{r}^{\mathsf{in}}) - 2^{m-1} \mod N$.
2: $(k'_0, k'_1) \leftarrow \mathsf{Gen}^{<}_m(1^\lambda, \mathsf{r}^{\mathsf{in}}, 1, \mathbb{S}_N)$.
3: For $b \in \{0,1\}$, let $k_b = k'_b || r_b$.
4: **return** (k_0, k_1).

$\mathsf{Eval}^{\mathsf{SExt}}_{m,n}(b, k_b, x)$:

1: Parse k_b as $k'_b || r_b$.
2: Set $x' \leftarrow x + 2^{m-1} \mod M$
3: Set $t \leftarrow M \cdot \mathsf{Eval}^{<}_m(b, k'_b, x')$.
4: **return** $u_b = b \cdot x' + r_b + t$.

Fig. 1. FSS Gate for Signed-Extension $\mathcal{G}_{\mathsf{SExt}}$, b refers to party id.

Theorem 2. *There is an FSS gate* $(\mathsf{Gen}^{\mathsf{SExt}}_{m,n}, \mathsf{Eval}^{\mathsf{SExt}}_{m,n})$ *for* $\mathcal{G}_{\mathsf{SExt}}$ *with keysize of* n *bits plus the keysize of* $\mathsf{DCF}_{m,\mathbb{S}_N}$ *and 1 invocation of* $\mathsf{DCF}_{m,\mathbb{S}_N}$ *in* $\mathsf{Eval}^{\mathsf{SExt}}_{m,n}$.

Proof. For $b \in \{0,1\}$, the simulator Sim_b for signed-extension gate is given x and u_b (the input and output of the ideal functionality). It first invokes Sim'_b, the simulator for DCF over m bits, which outputs a DCF key k'_b. It then computes x' and t in the same manner as Steps 2 and 3 in $\mathsf{Eval}^{\mathsf{SExt}}_{m,n}$ in Figure 1. Finally, it computes $r_b = u_b - b \cdot x' - t$ and outputs $k'_b || r_b$. One can easily see that the output of Sim_b is computationally indistinguishable from k_b, the output of $\mathsf{Gen}^{\mathsf{SExt}}_{m,n}$. □

Functionality	Function Notation	Description	FSS Gate	Key size per party	Online evaluation cost
Signed multiplication [7]	$g_{\times,n}(x,y)$; $x,y \in \mathbb{S}_N$	$x \cdot y \in \mathbb{S}_N$	$(\mathsf{Gen}_n^{\times}, \mathsf{Eval}_n^{\times})$	$3n$ bits	no FSS Eval calls
Matrix multiplication [7]	$g_{\times,n,d_1,d_2,d_3}(X,Y)$; matrices X,Y of dimension $d_1 \times d_2$ and $d_2 \times d_3$, with entries in $\mathbb{S}_N$	$X \cdot Y \in \mathbb{S}_N^{d_1 \times d_3}$	$(\mathsf{Gen}^{\times}_{n,d_1,d_2,d_3}, \mathsf{Eval}^{\times}_{n,d_1,d_2,d_3})$	$(d_1d_2 + d_2d_3 + d_1d_3)n$ bits	no FSS Eval calls
Interval containment [3]	$g_{\mathsf{IC},n,p,q}(x)$; $x,p,q \in \mathbb{S}_N$, $p \leqslant q$	$\mathbf{1}\{p \leqslant x \leqslant q\} \in \mathbb{S}_N$	$(\mathsf{Gen}^{\mathsf{IC}}_{n,p,q}, \mathsf{Eval}^{\mathsf{IC}}_{n,p,q})$	n bits + $\mathsf{DCF}_{n,\mathbb{S}_N}$	2 calls to $\mathsf{DCF}_{n,\mathbb{S}_N}$
Signed comparison [3]	$g_{\mathsf{sCMP},n}(x,y)$; $x,y \in \mathbb{S}_N$	$\mathbf{1}\{x \leqslant y\} \in \mathbb{S}_N$	$(\mathsf{Gen}_n^{\mathsf{sCMP}}, \mathsf{Eval}_n^{\mathsf{sCMP}})$	n bits + $\mathsf{DDCF}_{n-1,\mathbb{S}_N}$	1 call to $\mathsf{DDCF}_{n-1,\mathbb{S}_N}$
Arithmetic Right Shift (ARS) [3]	$g_{\gg_A,s,n}$; $x \in \mathbb{S}_N$, $s \in \mathbb{Z}$	$(x \gg_A s) = \frac{x - (x \mod 2^s)}{2^s} \in \mathbb{S}_N$	$(\mathsf{Gen}^{\gg_A}_{n,s}, \mathsf{Eval}^{\gg_A}_{n,s})$	n bits + $\mathsf{DCF}_{s,\mathbb{S}_N}$ + $\mathsf{DDCF}_{n-1,\mathbb{S}_N \times \mathbb{S}_N}$	1 call to $\mathsf{DCF}_{s,\mathbb{S}_N}$ and 1 call to $\mathsf{DDCF}_{n-1,\mathbb{S}_N \times \mathbb{S}_N}$
ReLU [3]	$g_{\mathsf{ReLU},n}(x)$; $x \in \mathbb{S}_N$	$x \cdot \mathbf{1}\{x \geqslant 0\} \in \mathbb{S}_N$	$(\mathsf{Gen}_n^{\mathsf{ReLU}}, \mathsf{Eval}_n^{\mathsf{ReLU}})$	$5n$ bits + $\mathsf{DCF}_{n,\mathbb{S}_N^2}$	2 calls to $\mathsf{DCF}_{n,\mathbb{S}_N^2}$
Splines [3]	$g_{\mathsf{spline},n,m,d,P,F}(x)$; $x \in \mathbb{S}_N$, $F = \{f_i\}_i$ is a set of m univariate polynomials of degree d with coefficients in $\mathbb{S}_N$, $P = \{p_0, p_1 \dots p_m\}$ is the list of $m+1$ knots in $\mathbb{S}_N$ with $p_0 = p_m = N-1$ and $p_i < p_{i+1}$ for $0 < i < m$.	$f_i(x) \in \mathbb{S}_N$ when $p_{i-1}+1 \leqslant x \leqslant p_i$	$(\mathsf{Gen}^{\mathsf{spline}}_{n,m,d,P}, \mathsf{Eval}^{\mathsf{spline}}_{n,m,d,P})$	$(2mn(d+1) + n)$ bits + $\mathsf{DCF}_{n,\mathbb{S}_N^{m(d+1)}}$	m calls to $\mathsf{DCF}_{n,\mathbb{S}_N^{m(d+1)}}$

Table 1. Description and costs for FSS gates from [3, 7].

3.2 Truncate-Reduce

The functionality of Truncate-Reduce (TR) for an n-bit number $x \in \mathbb{S}_N$ by s-bits is defined as dropping the last s bits and returning the result as a $(n-s)$-bit number $y \in \mathbb{S}_{2^{n-s}}$.

The $\mathcal{G}_{\mathsf{TR}}$ gate is the family of functions $g_{\mathsf{TR},n,s} : \mathbb{S}_N \to \mathbb{S}_{2^{n-s}}$ parameterized by input group $\mathbb{G}^{\mathsf{in}} = \mathbb{S}_N$ and output group $\mathbb{G}^{\mathsf{out}} = \mathbb{S}_{2^{n-s}}$, and defined as $g_{\mathsf{TR},n,s}(x) := x_{[s,n)}$. One straightforward way to realize the FSS gate for Truncate-Reduce is to use an FSS gate for arithmetic right shift operation that produces the result in n bits (see Table 1) followed by a local modulo operation to get rid of higher order s bits. In particular, $g_{\mathsf{TR},n,s}(x) = x \gg_A s \mod 2^{n-s}$, where $\gg_A$ represents the Arithmetic Right Shift (ARS) of x by s bits. This construction would have a total key size of n-bits plus the key size of $\mathsf{DCF}_{s,\mathbb{S}_N}$ and $\mathsf{DDCF}_{n-1,\mathbb{S}_N \times \mathbb{S}_N}$. In the following text, we provide a new construction that only uses a single DCF key, $\mathsf{DCF}_{s,2^{n-s}}$.

We denote the corresponding offset gate class by $\hat{\mathcal{G}}_{\mathsf{TR}}$ and the offset functions by:

$$\begin{aligned}\hat{g}^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{\mathsf{TR},n,s}(x) &= (x - \mathsf{r}^{\mathsf{in}})_{[s,n)} + \mathsf{r}^{\mathsf{out}} \mod 2^{n-s} \\ &= (x+y)_{[s,n)} + \mathsf{r}^{\mathsf{out}} \mod 2^{n-s}\end{aligned}$$

where $y = 2^n - \mathsf{r}^{\mathsf{in}}$. Using the relation from [30], we can re-write Truncate-Reduce as follows.

$$\begin{aligned}\hat{g}^{[\mathsf{r}^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{\mathsf{TR},n,s}(x) = x_{[s,n)} + y_{[s,n)} + \mathbf{1}\{x_{[0,s)} + y_{[0,s)} > 2^s - 1\} \\ + \mathsf{r}^{\mathsf{out}} \mod 2^{n-s}\end{aligned}$$

Based on this, we present our construction of the FSS gate for Truncate-Reduce in Figure 2 and summarize its cost below:

Truncate-Reduce Gate $(\mathsf{Gen}^{\mathsf{TR}}_{n,s}, \mathsf{Eval}^{\mathsf{TR}}_{n,s})$

$\mathsf{Gen}^{\mathsf{TR}}_{n,s}(1^\lambda, \mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:

1: Let $y = (2^n - \mathsf{r}^{\mathsf{in}}) \in \mathbb{S}_N$ and $\alpha^{(s)} = y_{[0,s)}$.
2: $(k_0^{(s)}, k_1^{(s)}) \leftarrow \mathsf{Gen}_s^{<}\left(1^\lambda, \alpha^{(s)}, 1, \mathbb{S}_{2^{n-s}}\right)$.
3: Sample random $r_0, r_1 \leftarrow \mathbb{S}_{2^{n-s}}$ s.t. $r_0 + r_1 = \mathsf{r}^{\mathsf{out}} + y_{[s,n)} \mod 2^{n-s}$.
4: **return** (k_0, k_1), where $k_b = k_b^{(s)} || r_b$ for $b \in \{0,1\}$.

$\mathsf{Eval}^{\mathsf{TR}}_{n,s}(b, k_b, x)$:

1: Parse k_b as $k_b^{(s)} || r_b$.
2: For $x^{(s)} = 2^s - x_{[0,s)} - 1$, $t_b \leftarrow \mathsf{Eval}_s^{<}(b, k_b^{(s)}, x^{(s)})$.
3: **return** $b \cdot x_{[n,s)} + r_b + t_b$.

Fig. 2. FSS Gate for Truncate-Reduce $\mathcal{G}_{\mathsf{TR}}$, b is party id.

Theorem 3. *There is an FSS gate* $(\mathsf{Gen}^{\mathsf{TR}}_{n,s}, \mathsf{Eval}^{\mathsf{TR}}_{n,s})$ *for* $\mathcal{G}_{\mathsf{TR}}$ *with key size of* $(n-s)$ *bits plus the key size of* $\mathsf{DCF}_{s,\mathbb{S}_{2^{n-s}}}$ *and 1 invocation of* $\mathsf{DCF}_{s,\mathbb{S}_{2^{n-s}}}$ *in* $\mathsf{Eval}^{\mathsf{TR}}_{n,s}$.

4 Linear Layers

Machine learning models that use low bitwidth fixed-point numbers to represent the model parameters, i.e., weights, as well as activations, inherently rely on very precise computation of intermediate operations [16, 17, 24]. For linear layers, this corresponds to values being multiplied and accumulated over high bitwidths, before being truncated to required output bitwidth. Below, we discuss our FSS protocols for such linear layers, starting with the basic operation of element-wise multiplication followed by matrix multiplications and convolutions. Since all our benchmarks use signed arithmetic, we focus on signed multiplications below. Unsigned operations are analogous and disusssed in Appendix A.

4.1 Signed multiplication

We define the signed multiplication gate $\mathcal{G}_{\mathsf{smult}}$ as the family of functions $g_{\mathsf{smult},m,n} : \mathbb{S}_M \times \mathbb{S}_N \to \mathbb{S}_L$ with $L = M \cdot N$ parameterized by input group $\mathbb{G}^{\mathsf{in}} = \mathbb{S}_M \times \mathbb{S}_N$ and output group $\mathbb{G}^{\mathsf{out}} = \mathbb{S}_L$, and given by $g_{\mathsf{smult},m,n}(a,b) := \mathsf{sint}_m(a) *_\ell \mathsf{sint}_n(b)$. Intuitively, this says that finite-bit signed values a and b are lifted to $\mathbb{Z}$ and multiplied (note that taking a modulo with L for $\ell = m+n$ does not loose any bits).

For $a' = a + 2^{m-1} \mod M$ and $b' = b + 2^{n-1} \mod N$, using Equation (1) we get:

$$\begin{aligned} g_{\mathsf{smult},m,n}(a,b) &= (a' - 2^{m-1}) \cdot (b' - 2^{n-1}) \mod L \\ &= a' \cdot b' - a' \cdot 2^{n-1} - b' \cdot 2^{m-1} + 2^{m+n-2} \mod L \end{aligned}$$

We denote the corresponding offset gate class by $\hat{\mathcal{G}}_{\mathsf{smult}}$ and the offset functions by:

$$\begin{aligned} &\hat{g}^{[\mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{in}}_2, \mathsf{r}^{\mathsf{out}}]}_{\mathsf{smult},m,n}(x,y) = g_{\mathsf{smult},m,n}(x - \mathsf{r}^{\mathsf{in}}_1, y - \mathsf{r}^{\mathsf{in}}_2) + \mathsf{r}^{\mathsf{out}} \\ &= ((x' - \mathsf{r}^{\mathsf{in}}_1) \mod M - 2^{m-1}) \cdot ((y' - \mathsf{r}^{\mathsf{in}}_2) \mod N \\ &- 2^{n-1}) + \mathsf{r}^{\mathsf{out}} \mod L \end{aligned}$$

where $x' = x + 2^{m-1} \mod M$ and $y' = y + 2^{n-1} \mod N$. We used Equation (1) in the above expression. Now, using Equation (2) we have:

$$\begin{aligned} \hat{g}^{[\mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{in}}_2, \mathsf{r}^{\mathsf{out}}]}_{\mathsf{smult},m,n}(x,y) &= \left(x' - \mathsf{r}^{\mathsf{in}}_1 - 2^{m-1} + 2^m \cdot \mathbf{1}\{x' < \mathsf{r}^{\mathsf{in}}_1\}\right) \\ &\cdot \left(y' - \mathsf{r}^{\mathsf{in}}_2 - 2^{n-1} + 2^n \cdot \mathbf{1}\{y' < \mathsf{r}^{\mathsf{in}}_2\}\right) + \mathsf{r}^{\mathsf{out}} \mod L \end{aligned}$$

We observe that the above relation requires two comparisons, one for x' with $\mathsf{r}^{\mathsf{in}}_1$ and second for y' with $\mathsf{r}^{\mathsf{in}}_2$. However, if we implement above relation naively with FSS it will require a 2-round protocol. The first round will compute shares of comparison outputs and the second round will do the multiplications (with shares of $\mathsf{r}^{\mathsf{in}}_2$ and $\mathsf{r}^{\mathsf{in}}_1$, resp.). Below, we provide a 1-round protocol for this gate using the observation that the values that need to be multiplied with $\mathbf{1}\{x' < \mathsf{r}^{\mathsf{in}}_1\}$ (resp., $\mathbf{1}\{y' < \mathsf{r}^{\mathsf{in}}_2\}$) are either known to the servers or the dealer. With this observation, we can increase the DCF payloads to 2 group elements each and compute the whole expression in a single round. First, we re-arrange the above expression to separate out the terms known to the servers and the dealer.

$$\begin{aligned} &\hat{g}^{[\mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{in}}_2, \mathsf{r}^{\mathsf{out}}]}_{\mathsf{smult},m,n}(x,y) = 2^m \cdot \mathbf{1}\{x' < \mathsf{r}^{\mathsf{in}}_1\} \cdot (y' - 2^{n-1}) \\ &+ 2^m \cdot \mathbf{1}\{x' < \mathsf{r}^{\mathsf{in}}_1\} \cdot (-\mathsf{r}^{\mathsf{in}}_2) + 2^n \cdot \mathbf{1}\{y' < \mathsf{r}^{\mathsf{in}}_2\} \cdot (x' - 2^{m-1}) \\ &+ 2^n \cdot \mathbf{1}\{y' < \mathsf{r}^{\mathsf{in}}_2\} \cdot (-\mathsf{r}^{\mathsf{in}}_1) + (-\mathsf{r}^{\mathsf{in}}_1) \cdot (y' - 2^{n-1}) \\ &+ (-\mathsf{r}^{\mathsf{in}}_2) \cdot (x' - 2^{m-1}) + \mathsf{r}^{\mathsf{in}}_1 \cdot \mathsf{r}^{\mathsf{in}}_2 + \mathsf{r}^{\mathsf{out}} \\ &+ (x' - 2^{m-1}) \cdot (y' - 2^{n-1}) \mod L \end{aligned}$$

Based on above re-arrangement, we present our construction of the FSS gate for Signed Multiplication in Figure 3 and summarize its cost below:

Theorem 4. *There is an FSS gate* $(\mathsf{Gen}^{\mathsf{smult}}_{m,n}, \mathsf{Eval}^{\mathsf{smult}}_{m,n})$ *for* $\mathcal{G}_{\mathsf{smult}}$ *which has a total keysize* $3(m+n)$ *bits plus the key size of* $\mathsf{DCF}_{m,\mathbb{S}^2_N}$ *and* $\mathsf{DCF}_{n,\mathbb{S}^2_M}$ *and requires 1 invocation each of* $\mathsf{DCF}_{m,\mathbb{S}^2_N}$ *and* $\mathsf{DCF}_{n,\mathbb{S}^2_M}$ *in* $\mathsf{Eval}^{\mathsf{smult}}_{m,n}$.

Remark. We use the above protocol for signed multiplication to realize element-wise multiplications occurring in Hadamard Product layers in our benchmarks.

4.2 Matrix Multiplication and Convolution

Consider signed matrix multiplication of $A \in \mathbb{S}_M^{d_1 \times d_2}$ and $B \in \mathbb{S}_N^{d_2 \times d_3}$. In the resulting matrix $C = A \cdot B$ each element is a result of d_2 multiplications and $d_2 - 1$ additions. Even if we store the result of multiplication in a larger ring $\mathbb{S}_{2^{m+n}}$, similar to signed multiplication, and do d_1 additions over these, the result can still overflow due to additions. To avoid such an overflow, underlying libraries assume that computation is happening over a sufficiently large domain. Note that $\ell = m + n + \lceil \log d_2 \rceil$ suffices to avoid any overflows during whole dot product computation. Even though the compute looks similar to what we discussed in last section, matrix multiplication

Signed Multiplication Gate $(\mathsf{Gen}^{\mathsf{smult}}_{m,n}, \mathsf{Eval}^{\mathsf{smult}}_{m,n})$

$\mathsf{Gen}^{\mathsf{smult}}_{m,n}(1^\lambda, \mathsf{r}^{\mathsf{in}}_1, \mathsf{r}^{\mathsf{in}}_2, \mathsf{r}^{\mathsf{out}})$:

1: Sample random $r_{10}, r_{11} \leftarrow \mathbb{S}_L$ s.t. $r_{10} + r_{11} = \mathsf{sint}_m(-\mathsf{r}^{\mathsf{in}}_1) \mod L$.
2: Sample random $r_{20}, r_{21} \leftarrow \mathbb{S}_L$ s.t. $r_{20} + r_{21} = \mathsf{sint}_n(-\mathsf{r}^{\mathsf{in}}_2) \mod L$.
3: Sample random $r_0, r_1 \leftarrow \mathbb{S}_L$ s.t. $r_0 + r_1 = \mathsf{sint}_m(\mathsf{r}^{\mathsf{in}}_1) \cdot \mathsf{sint}_n(\mathsf{r}^{\mathsf{in}}_2) + \mathsf{r}^{\mathsf{out}} \mod L$.
4: Let $\beta_1 = (1, -\mathsf{r}^{\mathsf{in}}_2) \in \mathbb{S}^2_N$ and $\beta_2 = (1, -\mathsf{r}^{\mathsf{in}}_1) \in \mathbb{S}^2_M$.
5: $(k_{10}, k_{11}) \leftarrow \mathsf{Gen}^{<}_m(1^\lambda, \mathsf{r}^{\mathsf{in}}_1, \beta_1, \mathbb{S}^2_N)$.
6: $(k_{20}, k_{21}) \leftarrow \mathsf{Gen}^{<}_n(1^\lambda, \mathsf{r}^{\mathsf{in}}_2, \beta_2, \mathbb{S}^2_M)$.
7: For $b \in \{0,1\}$, let $k_b = k_{1b}||k_{2b}||r_{1b}||r_{2b}||r_b$.
8: **return** (k_0, k_1).

$\mathsf{Eval}^{\mathsf{smult}}_{m,n}(b, k_b, x, y)$:

1: Parse $k_b = k_{1b}||k_{2b}||r_{1b}||r_{2b}||r_b$.
2: Set $x' = x + 2^{m-1} \mod M$, $y' = y + 2^{n-1} \mod N$.
3: Set $(t_1, t_2) \leftarrow \mathsf{Eval}^{<}_m(b, k_{1b}, x')$.
4: Set $(t_3, t_4) \leftarrow \mathsf{Eval}^{<}_n(b, k_{2b}, y')$.
5: Set $s_1 = 2^m \cdot (t_1 \cdot (y' - 2^{n-1}) + t_2)$
6: Set $s_2 = 2^n \cdot (t_3 \cdot (x' - 2^{m-1}) + t_4)$
7: **return** $s_1 + s_2 + r_{1b} \cdot (y' - 2^{n-1}) + r_{2b} \cdot (x' - 2^{m-1}) + r_b + b \cdot (x' - 2^{m-1}) \cdot (y' - 2^{n-1}) \mod L$.

Fig. 3. FSS Gate for Signed Multiplication $\mathcal{G}_{\mathsf{smult}}$, b is party id.

over mixed bitwidths cannot be computed in a single round, due to the need for a ring larger than $\mathbb{S}_{2^{m+n}}$. In particular, the element-wise multiplications of x and y performed during dot-product would also have a term $2^m\{x' < \mathsf{r}^{\mathsf{in}}_1\} \cdot 2^n\{y' < \mathsf{r}^{\mathsf{in}}_2\}$ that does not become 0 when taking a modulo over $L > 2^{m+n}$. Hence, we need to perform an explicit multiplication between 2 comparison outputs that are secret shared between the servers, and this leads to an additional round of interaction.

Since we are going to take 2 rounds, we use the approach of *extend-then-multiply*. For this, there are two ways. The first one follows the approach of [30] that extends the entries of one of the matrix by minimal amount, i.e., $\lceil \log d_2 \rceil$ then performs mixed bitwidth multiplication as discussed in previous section. However, unlike [30] this approach is quite expensive for FSS, as the payload of the DCF key for comparison used for a value, say a in matrix A, grows with number of elements in B it is multiplied with, i.e., d_3. Similarly, for elements in B, the payload of DCF key grows with d_1. So, say, we extend the entries of B from n to $n' = n + \lceil \log d_2 \rceil$ bits and then carry out a protocol similar to that of signed multiplication. Then, key would have size, roughly, $d_2 d_3(n' + \mathsf{DCF}_{n,\mathbb{U}_{2^{n'}}})$ for extensions plus $d_1 d_2 \mathsf{DCF}_{m,\mathbb{U}^{d_3+1}_{2^{n'}}}$ for comparisons on entries in A plus $d_2 d_3 \mathsf{DCF}_{n',\mathbb{U}^{d_1+1}_M}$ for comparisons on entries in sign-extended B plus $\ell(d_1 d_2 + d_2 d_3 + d_3 d_1)$ per party.

Now we describe the second approach that is much more efficient for FSS. In the first round, we sign extend entries of both A and B to $\ell = m + n + \lceil \log d_2 \rceil$-bits using Signed-Extension gate from Section 3.1. In the second round, we do uniform bitwidth matrix multiplication as in [7] (see Table 1). While the round complexity of this FSS protocol is still 2, the total keysize is only $d_1 d_2(\ell + \mathsf{DCF}_{m,\mathbb{U}_L}) + d_2 d_3(\ell + \mathsf{DCF}_{n,\mathbb{U}_L})$ for sign extensions plus $\ell(d_1 d_2 + d_2 d_3 + d_3 d_1)$ for matrix multiplication.

Convolutions. These can be directly reduced to matrix multiplications, but a trivial translation expands the input matrices and amounts to larger than necessary key size. To reduce keysize, following the ideas from [25], we sign extend and secret-share the input masks (as we do in uniform bitwidth matrix multiplication) for the input matrices instead of expanding and then calling signed extension and mask sharing.

Fixed-point mixed-bitwidth matrix multiplication. When operating over fixed-point arithmetic, the input matrices apart from the specified bitwidths also have respective scales, say, s_m and s_n. After following the above procedure of extend-then-multiply, the result is in bitwidth $\ell = m + n + \lceil \log d_2 \rceil$ and scale $s = s_m + s_n$. The fixed-point model would also specify the required bitwidth and scale for the output, say, $n_{\mathcal{O}}$ and $s_{\mathcal{O}}$. Now we adjust to this using appropriate truncation operations as follows: We can safely assume that $s_{\mathcal{O}} \leqslant s$ and $n_{\mathcal{O}} \leqslant \ell$. In the third round, we reduce the scale of output by $\mathsf{tr} = s - s_{\mathcal{O}}$ and adjust the bitwidth to $n_{\mathcal{O}}$. We have the following two cases:

- $n_{\mathcal{O}} \leqslant (\ell - \mathsf{tr})$: In this case, parties first compute a Truncate-Reduce gate to truncate and reduce by tr bits to obtain the result in $\ell - \mathsf{tr}$ bits and scale $s_{\mathcal{O}}$. Next, parties locally compute a modulo $2^{n_{\mathcal{O}}}$ to obtain output with correct bitwidth and scale.
- $n_{\mathcal{O}} > (\ell - \mathsf{tr})$: Here, parties compute an arithmetic right shift by tr bits to obtain the result in ℓ bits and scale $s_{\mathcal{O}}$. This is followed by a local modulo operation by $2^{n_{\mathcal{O}}}$ to obtain the output with correct bitwidth and scale.

Uniform bitwidth linear layers. For our benchmarks that have linear layers working over uniform bitwidth and scale, say n and s, (signed) arithmetic, we use the known matrix multiplication FSS gate from [7] to obtain

output in n-bits and scale $2s$ followed by truncation by s using arithmetic right shift gate from [3] to adjust the putput scale to s (see Table 1 to obtain the costs).

5 Math Functions

In this section, we first discuss our novel FSS-based protocols for *precise* math functions for the same input and output domains considered in [30]. To quantify how precise our math function implementations are, we use the standard notion of ULP error (defined formally in [14]) that we discuss below. Then, we provide a high-level overview for the design of our math functions, followed by mixed-bitwidth splines that are crucial to obtain *low-bitwidth* splines that are good approximations to math functions. Finally, we provide details on FSS-friendly math function design for popular activations – sigmoid and tanh – used in neural networks.

ULP error. It is impossible to represent an irrational value exactly using finite number of bits. Therefore, it is important to quantify the deviations between exact real result and the output of a math library in finite-bit representation. There are various notions of errors one can use – absolute, relative and ULP error. Standard math libraries use ULP error as a metric to determine whether the real output of a math function is *approximately* close to the finite-bit output that the library produces [1, 38]. The lower the ULP value, the higher is the precision and accuracy of the implementation of that math function. At a high level, ULP error between the exact real result r_1 and library output r_2 is equal to the number of representable values between r_1 and r_2 [14]. We use the same notion to quantify the precision of our math functions that use fixed-point as the finite-bit representation.

Design of math functions. Although our techniques are general, for a high level discussion, let us assume that we want to approximate sigmoid within 4 ULPs of error over fixed-point inputs and outputs with bitwidth 16 and scale 12. There are multiple design choices possible in coming up with such an implementation. For instance, SIRNN [30] used the recipe of first obtaining a good initial approximation followed by Goldsmidth's iterations where the number of iterations depend on final output scale precision desired. However, this approach leads to large number of online rounds and communication due to the iterative nature of the algorithm. Our first design choice is to use piecewise polynomials or *splines* to approximate math functions as these allow for one-round low communication protocols using FSS techniques [3, 7]. However, we notice that for obvious reasons, uniform bitwidth splines cannot be used to obtain low ULP errors. In particular, for the above mentioned case, we cannot find a reasonable spline that uses coefficients on 16-bits, does all arithmetic over 16-bits, and provides at most 4 ULP error for scales 12. Similar to [30], the polynomial computation in splines needs to happen with intermediate results in higher bitwidths and final result needs to be truncated (to reduce bitwidth and adjust scale). Here, intuitvely, while evaluating polynomials one keeps accumulating bitwidth and only reduces the final result.

We discuss details of our mixed-bitwidth splines followed by our protocols for precise math functions.

5.1 Mixed-bitwidth Splines

We first discuss the cleartext functionality for the mixed-bitwidth splines followed by their FSS implementation. For ease of exposition, we discuss the splines over integers (no scale) and later show to handle fixed-point arithmetic that has an associated scale with each value.

Suppose the spline under consideration is composed of m polynomials $f_1, f_2, \ldots, f_m$ (with degree d and coefficients of bitwidth n_c), and a set of $m+1$ knots P. Let the input to the spline be $x \in \mathbb{S}_{N_\mathcal{I}}$, where the bitwidth of x is $n_\mathcal{I}$ and $N_\mathcal{I} = 2^{n_\mathcal{I}}$. Let n be a sufficiently large bitwidth that prevents overflow of values during polynomial evaluation. Note that $n = n_c + d \cdot n_\mathcal{I}$ suffices for this purpose. The functionality $g^{\mathsf{mixed}}_{\mathsf{spline},(n_\mathcal{I},n_c),m,d,P,\{f_i\}_i} : \mathbb{S}_{N_\mathcal{I}} \to \mathbb{S}_N$ for mixed-bitwidth splines is defined as follows. First, sign extend the input x from $n_\mathcal{I}$ bits to n, resulting in $\mathsf{sint}(x) \mod N$. Then, sign extend the coefficients of all polynomials from n_c bits to n, and sign extend all knots from $n_\mathcal{I}$ bits to n. Finally output the result of uniform bitwidth spline functionality on input $\mathsf{sint}(x)$ using the new (sign extended) coefficients and knots. Note that this evaluation procedure (mod N) is the same as doing all computation over $\mathbb{Z}$.

Now we describe a simple (yet non-optimized) 2-round FSS-based protocol for this spline evaluation. For the first round, parties call the FSS gate for Signed-Extension $g_{\mathsf{SExt},n_\mathcal{I},n}$ with input $x \in \mathbb{S}_{N_\mathcal{I}}$ masked by r^{in}, and reconstruct $\bar{x} = \mathsf{sint}(x - \mathsf{r}^{\mathsf{in}}) + \mathsf{r}^{\mathsf{temp}} \mod N \in \mathbb{S}_N$. Here, $\mathsf{r}^{\mathsf{temp}} \in \mathbb{S}_N$ is chosen randomly by the dealer during key generation. For the second round, let $\bar{f}_i$ be the polynomial corresponding to f_i with (publicly known) coefficients sign extended to n-bits from n_c bits. Also, let $\bar{P}$ be the sign extended (publicly known) knots to n-bits.

Next, parties call the FSS gate for uniform-bitwidth splines $g_{\mathsf{spline},n,m,d,\bar{P},\{\bar{f}_i\}_i}$ from [3]. The input to this gate is $\bar{x} \in \mathbb{S}_N$ from the first round, masked by $\mathsf{r}^{\mathsf{temp}}$. The output would be the spline evaluated on $(\bar{x} - \mathsf{r}^{\mathsf{temp}})$ masked by $\mathsf{r}^{\mathsf{out}} \in \mathbb{S}_N$.

5.1.1 Optimizations

We propose two optimizations to the protocol described above that significantly reduce the FSS key size and offline and online computational cost.

Optimization 1. Here, we reduce the FSS key size for the spline gate used in the second round. In the above construction, the DCF key used in the spline operates over inputs from $\mathbb{S}_N$ (obtained after sign extending the original input) and uses the sign extended knots during the key generation by dealer and evaluation by servers. Our observation is as follows: Even though the parties need to learn the shares of coefficients of the correct polynomial to be evaluated in the larger domain, i.e., $\mathbb{S}_N$, the DCF input (and hence, its depth, etc) and knots themselves can come from the original domain $\mathbb{S}_{N_\mathcal{I}}$ where $N_\mathcal{I} = 2^{n_\mathcal{I}}$. In particular, we replace $\mathsf{DCF}_{n,\mathbb{S}_N^{m(d+1)}}$ in the above unoptimized scheme by $\mathsf{DCF}_{n_\mathcal{I},\mathbb{S}_N^{m(d+1)}}$ and evaluate it on the original input x instead of $\bar{x}$. This reduces the key size and the number of PRG calls made in key generation and evaluation of the uniform bitwidth spline gate (used in the second round of our protocol) by a factor of $n/n_\mathcal{I}$. For instance, for the case of sigmoid, this reduction factor is $4\times$.

Optimization 2. This optimization significantly reduces the number of PRG calls made by the servers during the online phase. Recall that the FSS gate for $g_{\mathsf{spline},n_\mathcal{I},m,d,P,\{\bar{f}_i\}_i}$ used in the second round uses a DCF with a payload of $m(d+1)n$ bits. This DCF key gets evaluated m times and the output of each invocation is $m(d+1)n$ bits. Let the output of the i^{th} invocation be $s_1^{(i)}, \ldots, s_m^{(i)}$ (using the same notation as Figure 5 in [3]). We observe that only $s_{i-1}^{(i)}, s_i^{(i)}$, i.e., $2(d+1)n$ bits, are used during evaluation and other values are discarded.

The $\mathsf{Gen}_n^<$ algorithm of the DCF construction of [3] generates a key k_b for each party $b \in \{0,1\}$ such that k_b consists of a random seed s_b and $n+1$ correction words $CW_i \in \mathbb{G}$ for $i \in \{1 \ldots n+1\}$ (where $\mathbb{G}$ denotes the output group). The seed generates a binary tree with 2^n leaves and each node is assosiated with a tuple (s_b, t_b, V_b) with an invariant that the sum of V_b along the evaluation path for an input x, form secret shares of $f^<_{\alpha,\beta}(x)$. Hence, in $\mathsf{Eval}_n^<$, it suffices to perform this addition along the respective path to get the desired output. In the case of splines, where $\mathbb{G}$ is a vector of group elements, these additions are performed element-wise. Since we need only 2 out of the m elements in the output of $\mathsf{Eval}_n^<$, we can tweak $\mathsf{Eval}_n^<$ to only do the required additions and PRG calls. This change reduces the number of PRG calls in the spline evaluation by roughly a factor of $m/2$. For the above example of sigmoid, this factor is roughly $10\times$.

Overall performance improvement. The two optimizations discussed above are compatible with each other and together lead to roughly a factor $n/n_\mathcal{I}$ reduction in FSS key size, $n/n_\mathcal{I}$ reduction in PRG calls by the dealer in key generation and $n/n_\mathcal{I} \cdot m/2$ reduction in PRG calls by the servers in the online phase. For the case of sigmoid, this amounts to $4\times$ reduction in key size, $4\times$ fewer PRG calls by the dealer and $40\times$ fewer PRG calls by the servers.

We now discuss how we can easily extend our protocol to work over fixed-point arithmetic as required.

5.1.2 Fixed-point arithmetic

In the context of fixed-point arithmetic, let the scale of the input x be $s_\mathcal{I}$, and that of the spline coefficients be s_c. In symbolic notation, let $f_i(x) = \sum_{j=0}^{d} a_{i,j} \cdot x^j$. The first round of our FSS protocol remains the same. Note that during polynomial evaluation in the spline, we require all the summands to have the same scale. This requires a small change in the dealer as follows: The dealer sign extends the coefficients from n_c to n bits and also left shifts $a_{i,j}$ by $(d-j) \cdot s_\mathcal{I}$ bits. So, during evaluation the scale of each summand of the polynomial is the same, viz. $s = s_c + d \cdot s_\mathcal{I}$. Now, after running the above described protocol, we have the output with bitwidth n and scale s.

Next, suppose the desired output is required to have bitwidth $n_\mathcal{O}$ and scale $s_\mathcal{O}$. We do this adjustment in the final round as follows: We can safely assume that $s_\mathcal{O} \leqslant s$ as to obtain precise output with scale $s_\mathcal{O}$, scales of the coefficients will have to be appropriately large as well. Now, in the third round, we reduce the scale of output by $\mathsf{tr} = s - s_\mathcal{O}$ and adjust the bitwidth to $n_\mathcal{O}$ by using appropriate truncation operations as discussed in "fixed-point mixed-bitwidth matrix multiplication" paragraph in Section 4.2. The complete 3-round FSS protocol is described in Appendix E.

Below, we summarize the key size and evaluation cost of the FSS protocol for mixed-bitwidth splines

over fixed-point $g^{(\mathsf{mixed,fixed})}_{\mathsf{spline},(n_\mathcal{I},s_\mathcal{I},n_\mathcal{O},s_\mathcal{O},n_c,s_c),m,d,P,\{f_i\}_i}$. (Let $\mathcal{G}_{(\mathsf{mixed,fixed})\text{-}\mathsf{spline}}$ denote the corresponding function family, parameterized accordingly). After optimization 2, our protocol for spline evaluation uses the underlying DCF key in a non-black box manner, hence we report the cost of this step in number of PRG calls made. Also, we report the cost when the third round of the protocol uses a Truncate-Reduce gate. The other case is similar.

Theorem 5. *Let* $\mathsf{params} = (n_\mathcal{I}, s_\mathcal{I}, n_\mathcal{O}, s_\mathcal{O}, n_c, s_c)$, $n = n_c + d\cdot n_\mathcal{I}$, $\mathsf{tr} = s_c + d\cdot s_\mathcal{I} - s_\mathcal{O}$. *There is a 3-round FSS protocol* $(\mathsf{Gen}^{(\mathsf{mixed,fixed})\text{-}\mathsf{spline}}_{\mathsf{params},m,d,P}, \mathsf{Eval}^{(\mathsf{mixed,fixed})\text{-}\mathsf{spline}}_{\mathsf{params},m,d,P})$ *for mixed-bitwidth splines over fixed-point that has a total key size of* $2mn(d+1)+n$ *bits, plus the key size of* $\mathsf{DCF}_{n_\mathcal{I},\mathbb{S}_N^{m(d+1)}}$, *plus the key sizes of FSS gates for* $g_{\mathsf{SExt},n_\mathcal{I},n}$ *and* $g_{\mathsf{TR},n,\mathsf{tr}}$. *Let* $\ell = \lceil \frac{2n(d+1)}{4\lambda+2} \rceil$, *where* λ *is the security parameter. The online phase makes single evaluations of Sign-Extension and Truncate-Reduce gates and at most* $m(n_\mathcal{I}(1+\ell)+\ell)$ *calls to PRG G (used in DCF) during spline evaluation (in the second round).*

5.2 Math Functions

In this section, we discuss our approach for computing math functions using FSS techniques – in particular, sigmoid and tanh. We use mixed-bitwidth splines over fixed-point as approximations to math functions that can be realized directly using the 3-round protocol described in the previous section. Below, we discuss how we obtain the required splines for each of the math functions. We use sigmoid to illustrate this.

Sigmoid. Over the reals, $\mathsf{sigmoid}(x) = 1/(1+e^{-x})$ and tends to 0 for small values of x and tends to 1 for large values of x. Our task is as follows: Given the bitwidths and scales for the inputs and outputs, find a spline that approximates the real result with at most 4ULPs of error (see beginning of this section for the definition of ULP error). The first step is to *clip* the input domain to an interesting interval as follows: we find the largest x_L and the smallest x_R such that if we set $\mathsf{sigmoid}(x) = 0$ (with appropriate fixed-point representation of outputs) for all $x \leqslant x_L$ and set $\mathsf{sigmoid}(x) = 1$ for all $x \geqslant x_R$, the resulting ULP error $\leqslant 4$. In the second step, we start with a choice of degree of the polynomials, d, and number of knots, m, and run an off-the-shelf tool Octave [12] to find a best fit spline for sigmoid for the reduced domain. Note that this step, returns a floating-point spline, i.e., both polynomial coefficients as well as knots are floating-point values.

In the third step, we quantize this spline, i.e., represent it over fixed-point as follows: we quantize the knots to have the same bitwidth and scale as our inputs. We linearly search over bitwidths and scales for the coefficients. For a choice of n_c, s_c, we exhaustively run the mixed-bitwidth spline cleartext algorithm for all inputs and check for their ULP error w.r.t. the output of a high-precision math library [13]. We crucially note that since sigmoid (and also tanh) are well-behaved functions with bounded outputs, and the output scale $s_\mathcal{O} \leqslant 14$, this exhaustive testing is feasible. If the maximum ULP error $\leqslant 4$ we stop. Otherwise, we increase the value of either n_c or s_c until $n_c = 32$. If we do not find a good approximation, we increase the number of knots, m, until 100; even if this is unsuccessful, we increment the degree d and go back to the second step of spline finding.

Following the above procedure, we successfully find splines with $d = 2, m \leqslant 52$ for the sigmoid function for input and output scales such that $0 \leqslant s_\mathcal{I}, s_\mathcal{O} \leqslant 14$ and this suffices for our benchmarks as well as benchmarks considered in prior works. As is expected from a 2D graph of sigmoid, we were unable to find linear splines for sigmoid (and also tanh) with even 100 knots.

Tanh. Over the reals, $\mathsf{tanh}(x) = (e^x - e^{-x})/(e^x + e^{-x})$ and tends to -1 for small values of x and 1 for large values. Our procedure for tanh is identical to sigmoid except for a straightword change to clipping in terms of outputs on inputs with large magnitude.

Reciprocal square root. Over reals, $\mathsf{rsqrt}(x) = 1/\sqrt{x}, x > 0$. To avoid division-by-zero error when x is very small, we assume that all inputs to rsqrt satisfy $x \geqslant \epsilon$, where ϵ is a small public constant. As in the case of SIRNN [30], we set $\epsilon = 0.1$. The procedure to find splines is similar to sigmoid and tanh with one difference. We observe that since the precision of input x is $s_\mathcal{I}$, it suffices to compute the output with precision $s_\mathcal{I}/2$. Hence, the ULP error of spline obtained is computed over bitwidth $n_\mathcal{O}$ and scale $\lceil s_\mathcal{I}/2 \rceil$, instead of $s_\mathcal{O}$. Later, we adjust the scale of spline output to $s_\mathcal{O}$ by left-shifting the output by $(s_\mathcal{O} - \lceil s_\mathcal{I}/2 \rceil)$ bits.

Sample choice of parameters. Table 2 lists the choice of our spline parameters that give at most 4 ULP error for various configurations of math functions required by our benchmarks in Section 6.2. In Appendix B, we provide fixed-point values of the coefficients and intervals of a mixed-bitwidth spline for tanh.

Function	$n_{\mathcal{I}} = n_{\mathcal{O}}$	$s_{\mathcal{I}}$	$s_{\mathcal{O}}$	d	m
Sigmoid ($n_c = 32, s_c = 20$)	16	8	14	2	34
	16	9	14	2	34
	16	11	14	2	34
	16	13	14	2	29
	16	12	12	2	19
	37	12	12	2	20
Tanh ($n_c = 32, s_c = 18$)	16	8	8	2	10
	16	9	9	2	12
	16	11	11	2	20
	16	12	12	2	26
	16	13	13	2	12
	37	12	12	2	26
Reciprocal square root ($n_c = 32, s_c = 13$)	16	10	9	2	10
	16	12	11	2	10

Table 2. Spline parameters (degree d, number of intervals m, coefficient bitwidth n_c, coefficient scale s_c) with at most 4 ULPs error, for varying input bitwidth $n_{\mathcal{I}}$, output bitwidth $n_{\mathcal{O}} = n_{\mathcal{I}}$, input scale $s_{\mathcal{I}}$, output scale $s_{\mathcal{O}}$.

6 Evaluation

In this section, we perform a empirical evalutation of LLAMA and compare its performance with relevant prior works. In Section 6.1, we provide microbenchmarks that compare our protocols for mixed-bitwidth arithmetic and math functions with SIRNN [30] and MP-SPDZ [23], which are the prior state-of-the-art systems for precise implementations of these functions. SIRNN is a 2PC system in the semi-honest setting and MP-SPDZ is run in the semi-honest 2PC setting with trusted dealer. (SIRNN is optimized for end-to-end latency, while MP-SPDZ, like LLAMA, considers an offline-online split.) For these microbenchmarks, LLAMA reduces the online communication by two orders of magnitude and latency by $1.9 - 10\times$.

Next, in Section 6.2, we use LLAMA to run end-to-end secure inference on various neural network benchmarks and compare its performance with appropriate prior works that considered same benchmarks. We evaluate on the benchmarks considered in SIRNN that use math functions and/or mixed-bitwidth computations. We observe that online latency and communication using LLAMA is up to $57\times$ and $12000\times$ lower than SIRNN. To demonstrate scalability of LLAMA, we evaluate it on large convolutional networks such as ResNet-50 for ImageNet and contrast its performance with recent systems such as CrypTFlow [25] for 3PC and CrypTFlow2 [31] for 2PC. We also compare with DELPHI [28], a 2PC work that explicitly considers the question of online latency of 2-party secure inference. Finally, we compare LLAMA with AriaNN [34], an FSS-based secure inference framework (with erroneous ReLU and truncations and no support for math functions or mixed-bitwidth operations), on a benchmark considered in that work.

Implementation Details. LLAMA is implemented as a C++ library with ~6700 lines of code. The code is publically available at https://github.com/mpc-msri/EzPC/tree/master/FSS. In addition to the FSS protocols for mixed-bitwidth and math functions, we also implement the relevant FSS schemes and gates proposed in [3], as their work does not have an implementation. All APIs for key generation and evaluation are parameterized by input and output bitwidths and scales, to easily support both uniform and mixed-bitwidth operations. As suggested in [39, Section 6], we use the Matyas-Meyer-Oseas one-way compression function (which uses AES in fixed-key mode) to generate pseudorandomness. This is done to avoid multiple expensive AES initialization operations in the DCF computation, which is the main cost of FSS protocols.

We have integrated LLAMA as a cryptographic backend to EzPC [8]. This allows us to compile fixed-point inference code written in EzPC, into FSS-friendly C++ code. Various frontends such as CrypTFlow [25] and SeeDot [17] can be easily used to obtain fixed-point EzPC code (with carefully chosen bitwidths and scales that preserve accuracy) for arbitrary machine learning network architectures.

Experimental Setup. We run our benchmarks on 3 virtual machines (one dealer and 2 servers), each with a 4-core 3.7 GHz Xeon processor and 16 GBs of RAM. In the LAN setting, all VMs are connected in a network with average bandwidth of 340 MBps and RTT of 0.96 ms, while in the WAN setting the corresponding numbers are 120 MBps and 72.3 ms respectively. The mean and standard deviation of both offline and online execution times are calculated over 100 runs.

6.1 Microbenchmarks

In this section we microbenchmark LLAMA on individual functions used in mixed-bitwidth arithmetic as well as math functions. For precise math functions, we compare with SIRNN [30] and MP-SPDZ [23]. Although SIRNN is a standard 2PC system and LLAMA is a 2PC system in the dealer model, SIRNN is the only work that considers secure computation of mixed-bitwidth operations, and hence we compare with it for these blocks. Microbenchmarks for bitwidth changing functions, i.e., signed-extension and truncate-reduce, and math functions, i.e., sigmoid, tanh and reciprocal square root are provided in Table 3, while those for mixed-bitwidth matrix multiplication are presented in Table 4. For Ta-

Layer	Batch Size	Technique	Communication (in KB)		Online Rounds	LAN (in milliseconds)	
			Offline	Online		Offline	Online
Signed-Extension ($m = 8, n = 21$)	100	LLAMA	35	0.8	1	0.38 ± 0.14	0.37 ± 0.26
		SIRNN	-	30	7	-	4.5 ± 0.6
	1000	LLAMA	352	7.8	1	0.96 ± 0.68	0.81 ± 0.22
		SIRNN	-	114	7	-	5.73 ± 1.54
Truncate-Reduce ($n = 21, s = 13$)	100	LLAMA	47	0.2	1	0.47 ± 0.49	0.48 ± 0.43
		SIRNN	-	41	13	-	9.34 ± 6.34
	1000	LLAMA	466	2	1	1.72 ± 1.91	0.89 ± 0.44
		SIRNN	-	211	13	-	13.65 ± 2.17
Sigmoid ($n_{\mathcal{I}} = n_{\mathcal{O}} = 16, s_{\mathcal{I}} = 9, s_{\mathcal{O}} = 14$)	100	LLAMA	3297	3.5	3	12.47 ± 5.5	4.09 ± 1.47
		SIRNN	-	768	139	-	91.96 ± 8.50
		MP-SPDZ	3696	134	145	**	32.32 ± 8.12
	1000	LLAMA	33044	35	3	128.45 ± 46.91	27.05 ± 4.37
		SIRNN	-	5007	139	-	102.46 ± 8.06
		MP-SPDZ	5246	1308	145	**	52.10 ± 8.90
Tanh ($n_{\mathcal{I}} = n_{\mathcal{O}} = 16, s_{\mathcal{I}} = s_{\mathcal{O}} = 9$)	100	LLAMA	1320	3.5	3	5.35 ± 3.88	2.81 ± 0.84
		SIRNN	-	604	131	-	83.7 ± 8.26
		MP-SPDZ	3696	137	155	**	35.74 ± 12.46
	1000	LLAMA	13219	35	3	51.06 ± 19.99	10.16 ± 3.44
		SIRNN	-	3614	131	-	88.07 ± 8.96
		MP-SPDZ	5246	1341	155	**	57.60 ± 8.80
Reciprocal square root ($n_{\mathcal{I}} = n_{\mathcal{O}} = 16, s_{\mathcal{I}} = 12, s_{\mathcal{O}} = 11$)	100	LLAMA	1138	3.5	3	4.80 ± 4.35	2.84 ± 1.10
		SIRNN	-	881	185	-	124.05 ± 10.95
		MP-SPDZ	2457	44.4	87	**	22.11 ± 5.00
	1000	LLAMA	11375	35	3	41.20 ± 15.60	8.99 ± 1.79
		SIRNN	-	5488	185	-	126.03 ± 11.41
		MP-SPDZ	2467	413	87	**	28.92 ± 5.78

Table 3. Performance comparison for bitwidth changing and math functions. For Signed-Extension, m,n are input, output bitwidths. For Truncate-Reduce, n is input bitwidth and s is shift amount. For Sigmoid, Tanh and Reciprocal square root, $n_{\mathcal{I}}, n_{\mathcal{O}}, s_{\mathcal{I}}, s_{\mathcal{O}}$ denote input/output bitwidths and scales. ** denotes that the value was not reported by the code.

ble 3, the choice of parameters for bitwidths and scales are made using examples from our benchmarks such as Google-30 [26] and Heads [30] (described in Section 6.2) and we evaluate for batch sizes of 100 and 1000. For the math functions, for these choice of bitwidths and scales, Table 2 provides details on the spline chosen by LLAMA, i.e., degree and number of knots, as well as coefficient bitwidths and scales. From Table 3, LLAMA is up to 40× better than MP-SPDZ in online communication, up to 51× better in terms of online rounds, and up to 12× better in online execution time. As seen in Table 3, LLAMA communicates between $105 - 251\times$ less than SIRNN in the online phase and has between $13-61\times$ fewer rounds of online communication. In terms of performance, LLAMA is between $3.7-43\times$ faster than SIRNN in the LAN setting. Finally, as expected, while the total communication of LLAMA (i.e. communication including the offline key size as well) can be comparable to SIRNN in a few cases (e.g. Truncate-Reduce), LLAMA does have a larger total communication (by up to 4.4×) in other cases.

Table 4 summarizes our microbenchmarks for mixed-bitwidth matrix multiplication (multiplying a $d_1 \times d_2$ matrix with a $d_2 \times d_3$ matrix). The input/output bitwidths for all experiments are 8, while the scale is 6; however, due to d_2 being different in each case, the intermediate bitwidth ($16 + \lceil \log d_2 \rceil$) in each computation is different. As can be seen from the table, LLAMA communicates between $59 - 208\times$ less than SIRNN in the online phase and has $10 - 13\times$ fewer rounds. LLAMA also performs between $2.2-7.3\times$ better in the LAN setting. Further, in these microbenchmarks, LLAMA also has $1.3 - 5.4\times$ lower total communication.

6.2 Benchmarks

In this section, we evaluate and compare the performance of LLAMA on several machine learning inference

d_1	d_2	d_3	Technique	Communication (in MB)		Online Rounds	LAN (in milliseconds)	
				Offline	Online		Offline	Online
10	200	1000	LLAMA	72.44	1.63	3	316.81 ± 82.11	106.08 ± 4.71
			SIRNN	-	97.51	31	-	239.96 ± 15.02
10	2000	100	LLAMA	76.60	1.68	3	333.78 ± 86.22	110.91 ± 5.25
			SIRNN	-	108.36	39	-	266.19 ± 15.37
200	200	200	LLAMA	37.08	0.99	3	158.94 ± 46.24	56.63 ± 3.97
			SIRNN	-	206.87	41	-	415.63 ± 25.86

Table 4. Comparison for mixed-bitwidth matrix multiplication for dimensions $d_1 \times d_2$ and $d_2 \times d_3$, using bitwidths of 8 and scale 6.

algorithms. We provide details on the benchmarks considered in Appendix D and summarize the findings in Table 5. We split the discussion below into two kinds of benchmarks. Our main focus is the networks with math functions or networks that use low bitwidths for activations and weights for efficiency. We also consider simple convolutional neural networks (CNNs) from prior works to demonstrate our generality and scalability.

Neural networks with math functions/mixed-bitwidth arithmetic. First, to illustrate the performance of LLAMA on algorithms that use mixed-bitwidth arithmetic and/or math functions (tanh, sigmoid, or reciprocal square root), we run it on the following end-to-end inference benchmarks: DeepSecure B4 [32], that enables embedded sensors to classify various physical activities, as well as on an RNN algorithm [26] that enables keyword spotting on the Google-30 dataset [40]. We compare the performance of LLAMA with SIRNN and observe that LLAMA has up to 4 orders of magnitude lower online communication, up to 22× fewer online rounds, and up to 57× faster runtime. We also evaluate LLAMA on the sigmoid/tanh layers of the MiniONN LSTM [27] (a language model for word predictions) and the Industrial-72 benchmarks [24, 30] (a model that provides feedback for quality of shots in a sports game), as well as the reciprocal square root layers from the Heads model [35] (a model for counting the number of people in an image). Here, we show that, in comparison with SIRNN, the online communication of LLAMA is at least 200× less, the number of rounds is at least 43× better and the performance is at least 15× and 43× better in the LAN and WAN settings.

Other neural networks. While not the primary focus of this work, for the sake of completeness, we also compare LLAMA with prior systems on neural networks not requiring mixed-bitwidth arithmetic or math functions. We compare with DELPHI [28] – a 2PC system designed speifically with online cost in mind – and show $\approx 24\times$ better communication and $\approx 3-5\times$ better runtime for online phase. To illustrate that LLAMA can scale to large benchmarks, we run it on the ResNet-50 CNN on the ImageNet dataset [18], and compare with both CrypTFlow (a 3PC system) as well as CrypTFlow2 (a 2PC system)[6]. Finally, we consider AriaNN [34], which like LLAMA is an FSS based secure inference system (in the trusted dealer model), but does not support mixed-bitwidth arithmetic or math functions. Here alone, since AriaNN code [33] does not support execution on different VMs, we ran all parties in both AriaNN and LLAMA on the same VM and appropriately set the latency and bandwidth on the VM using the `tc` command[7]. On the ResNet-18 benchmark on Hymenoptera dataset, we show that LLAMA outperforms AriaNN by about 3× in online communication and 1.7× in online runtime (despite AriaNN using a probabilistically correct, cheaper, local truncation protocol compared to the correct truncation in LLAMA). This improvement can be attributed to ReLU being a 2-round protocol in AriaNN compared to an FSS gate in LLAMA. For fairness, we also provide numbers for LLAMA with the same probabilistically correct local truncation.

7 Conclusion

This paper proposes LLAMA, an FSS-based 2PC secure inference system in the semi-honest, trusted dealer setting. The main design goal of LLAMA is to minimize

6 In very recent work, Cheetah [19] show an improvement of ≈ 12× in communication and 4-5× in runtime over CrypTFlow2. We do not directly compare with this work, as it is orthogonal to the focus of this work; however, even in comparison to Cheetah, we note that LLAMA has much lower communication and is expected to outperform it.

7 The end-to-end code execution time in AriaNN took around 40 minutes. In Table 5, we report the offline and online times (around 350 seconds and 13 seconds respectively) that is output by their code. Due to longer execution times, the mean and standard deviation of runtimes are calculated over 25 iterations.

Network	Technique	Communication (in MB)		Online Rounds	LAN Time (in seconds)		WAN Time (in seconds)	
		Offline	Online		Offline	Online	Offline	Online
DeepSecure B4	LLAMA	183	0.15	21	0.78 ± 0.20	0.11 ± 0.01	3.15 ± 0.16	0.83 ± 0.05
	SIRNN	-	1844	379	-	6.45 ± 0.31	-	47.63 ± 1.67
Google-30	LLAMA	882	8.6	2687	5.31 ± 0.94	1.89 ± 0.12	8.02 ± 0.32	87.71 ± 3.65
	SIRNN	-	415	59899	-	37.18 ± 1.95	-	1997.8 ± 93.5
MiniONN LSTM (only Sigmoid, Tanh)	LLAMA	49.7	0.04	6	0.21 ± 0.07	0.02 ± 0.003	0.49 ± 0.23	0.23 ± 0.01
	SIRNN	-	9.7	403	-	0.34 ± 0.02	-	14.47 ± 0.90
Industrial-72 (only Sigmoid, Tanh)	LLAMA	19.4	0.03	42	0.09 ± 0.03	0.04 ± 0.006	0.30 ± 0.04	1.41 ± 0.07
	SIRNN	-	7.9	1847	-	1.23 ± 0.08	-	61.36 ± 3.85
Heads (only Reciprocal square root)	LLAMA	30	0.09	9	0.11 ± 0.03	0.026 ± 0.003	0.50 ± 0.02	0.23 ± 0.01
	SIRNN	-	18	545	-	0.39 ± 0.02	-	19.17 ± 0.77
MiniONN CNN	LLAMA	1084	8.2	25	4.61 ± 1.12	0.52 ± 0.02	10.05 ± 2.77	1.77 ± 0.10
	DELPHI	3258	196	**	36.73 ± 0.70	2.73 ± 0.07	55.63 ± 2.33	5.29 ± 0.25
	CrypTFlow2	-	340	345	-	10.24 ± 0.17	-	32.50 ± 2.50
ResNet-50 (ImageNet)	LLAMA	78848	745	280	427.94 ± 45.32	36.99 ± 0.46	631.91 ± 31.60	108.68 ± 7.17
	CrypTFlow2	-	31502	4053	-	476.6 ± 2.51	-	1026.9 ± 70.2
	CrypTFlow	-	6549	>7400	-	58.55 ± 3.46	-	364.7 ± 10.77
ResNet-18 (Hymenoptera)	LLAMA	8459	57	66	36.28 ± 10.29	7.29 ± 0.36	84.51 ± 3.23	12.79 ± 0.28
	LLAMA (local truncation)	5243	45	48	22.55 ± 6.00	6.81 ± 0.78	51.09 ± 1.29	9.28 ± 0.3
	AriaNN	6702	148	**	339.02 ± 18.51	12.27 ± 1.02	455.12 ± 18.76	55.62 ± 2.79

Table 5. Secure inference benchmarks using LLAMA and prior works. ** denotes that the value was not reported by the code.

online complexity. LLAMA proposes novel FSS-based constructions for signed-extension and truncate-reduce, which facilitate mixed-bitwidth operations and precise math functions based on low bitwidth splines. Due to the emphasis on lower online complexity, the offline phase in LLAMA can incur significant memory and bandwidth requirement (primarily attributed to large size of DCF keys). It would be interesting to optimize memory usage in the offline phase in LLAMA.

8 Acknowledgement

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. We thank Rahul Sharma for his help in spline error calculation and Deevashwer Rathee, Mayank Rathee and Rahul Kranti Kiran Goli for their help in understanding existing code bases.

References

[1] Intel SVML. https://software.intel.com/content/www/us/en/develop/documentation/mkl-vmperfdata/top.html (2022)
[2] Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO (1991)
[3] Boyle, E., Chandran, N., Gilboa, N., Gupta, D., Ishai, Y., Kumar, N., Rathee, M.: Function secret sharing for mixed-mode and fixed-point secure computation. In: EUROCRYPT (2020)
[4] Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Scholl, P.: Efficient pseudorandom correlation generators: Silent OT extension and more. In: CRYPTO (2019)
[5] Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: EUROCRYPT (2015)
[6] Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: Improvements and extensions. In: CCS (2016)
[7] Boyle, E., Gilboa, N., Ishai, Y.: Secure computation with preprocessing via function secret sharing. In: TCC (2019)
[8] Chandran, N., Gupta, D., Rastogi, A., Sharma, R., Tripathi, S.: EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In: IEEE EuroS&P (2019)

[9] Dalskov, A.P.K., Escudero, D., Keller, M.: Secure evaluation of quantized neural networks. Proc. Priv. Enhancing Technol. (2020)
[10] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO (2012)
[11] Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition (2009)
[12] Eaton, J.W., Bateman, D., Hauberg, S., Wehbring, R.: GNU Octave version 6.1.0 manual: a high-level interactive language for numerical computations (2020), https://www.gnu.org/software/octave/doc/v6.1.0/
[13] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. (2007)
[14] Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. (1991)
[15] Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: STOC (1987)
[16] Google: Tensorflow Lite (2019), https://www.tensorflow.org/lite/
[17] Gopinath, S., Ghanathe, N., Seshadri, V., Sharma, R.: Compiling kb-sized machine learning models to tiny iot devices. In: PLDI (2019)
[18] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016)
[19] Huang, Z., jie Lu, W., Hong, C., Ding, J.: Cheetah: Lean and fast secure two-party deep neural network inference. In: USENIX Security (2022)
[20] Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In: USENIX Security (2018)
[21] Kaissis, G., Ziller, A., Passerat-Palmbach, J., Ryffel, T., Usynin, D., Trask, A., Lima, I., Mancuso, J., Jungmann, F., Steinborn, M., Saleh, A., Makowski, M.R., Rueckert, D., Braren, R.: End-to-end privacy preserving deep learning on multi-institutional medical imaging. Nat. Mach. Intell. (2021)
[22] Kamara, S., Mohassel, P., Raykova, M., Sadeghian, S.S.: Scaling private set intersection to billion-element sets. In: Christin, N., Safavi-Naini, R. (eds.) FC (2014)
[23] Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: ACM CCS (2020)
[24] Kumar, A., Seshadri, V., Sharma, R.: Shiftry: RNN Inference in 2KB of RAM. In: OOPSLA (2020)
[25] Kumar, N., Rathee, M., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: CrypTFlow: Secure TensorFlow Inference. In: IEEE S&P (2020)
[26] Kusupati, A., Singh, M., Bhatia, K., Kumar, A., Jain, P., Varma, M.: FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network. In: NeurIPS (2018)
[27] Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minionn transformations. In: CCS (2017)
[28] Mishra, P., Lehmkuhl, R., Srinivasan, A., Zheng, W., Popa, R.A.: DELPHI: A cryptographic inference service for neural networks. In: USENIX Security (2020)
[29] Mohassel, P., Zhang, Y.: SecureML: A system for scalable privacy-preserving machine learning. In: IEEE S&P (2017)
[30] Rathee, D., Rathee, M., Goli, R.K.K., Gupta, D., Sharma, R., Chandran, N., Rastogi, A.: SIRNN: A math library for secure RNN inference. In: IEEE S&P (2021)
[31] Rathee, D., Rathee, M., Kumar, N., Chandran, N., Gupta, D., Rastogi, A., Sharma, R.: CrypTFlow2: Practical 2-party secure inference. In: CCS (2020)
[32] Rouhani, B.D., Riazi, M.S., Koushanfar, F.: Deepsecure: Scalable provably-secure deep learning. In: DAC (2018)
[33] Ryffel, T., Pointcheval, D., Bach, F.: ARIANN: Low-interaction privacy-preserving deep learning via function secret sharing. https://github.com/LaRiffle/ariann (2020)
[34] Ryffel, T., Pointcheval, D., Bach, F.: ARIANN: Low-interaction privacy-preserving deep learning via function secret sharing. PoPETS (2022)
[35] Saha, O., Kusupati, A., Simhadri, H.V., Varma, M., Jain, P.: RNNPool: Efficient non-linear pooling for RAM constrained inference. In: NeurIPS (2020)
[36] Soin, A., Bhatu, P., Takhar, R., Chandran, N., Gupta, D., Alvarez-Valle, J., Sharma, R., Mahajan, V., Lungren, M.P.: Production-level open source privacy preserving inference in medical imaging. CoRR (2021)
[37] Vadapalli, A., Bayatbabolghani, F., Henry, R.: You may also like... privacy: Recommendation systems meet pir. In: PoPETs (2021)
[38] Wang, E., Zhang, Q., Bo, S., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel math kernel library (2014)
[39] Wang, F., Yun, C., Goldwasser, S., Vaikuntanathan, V., Zaharia, M.: Splinter: Practical private queries on public data. In: Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (2017)
[40] Warden, P.: Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. arXiv (2018)
[41] Yao, A.C.: How to generate and exchange secrets. In: FOCS (1986)

A Unsigned multiplication

Unsigned Multiplication of two values $x \in \mathbb{U}_M, y \in \mathbb{U}_N$ refers to the multiplication of the two values $\mathsf{uint}_m(x)$ and $\mathsf{uint}_n(y)$ carried out in the group $\mathbb{U}_L$, where $L = M \cdot N$, which is equivalent to $\mathsf{uint}_m(x) *_\ell \mathsf{uint}_n(y)$.

The unsigned multiplication gate $\mathcal{G}_{\mathsf{umult}}$ is the family of functions $g_{\mathsf{umult},m,n} : \mathbb{U}_M \times \mathbb{U}_N \to \mathbb{U}_L$ parameterized by input group $\mathbb{G}^{\mathsf{in}} = \mathbb{U}_M \times \mathbb{U}_N$ and output group $\mathbb{G}^{\mathsf{out}} = \mathbb{U}_L$, and given by $g_{\mathsf{umult},m,n}(x, y) := \mathsf{uint}_m(x) *_\ell \mathsf{uint}_n(y)$.

We denote the corresponding offset gate class by $\hat{\mathcal{G}}_{\mathsf{umult}}$ and the offset functions by

$$\begin{aligned}\hat{g}^{[\mathsf{r}_1^{\mathsf{in}},\mathsf{r}_2^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{\mathsf{umult},m,n}(x,y) &= g_{\mathsf{umult},m,n}(x-\mathsf{r}_1^{\mathsf{in}}, y-\mathsf{r}_2^{\mathsf{in}}) + \mathsf{r}^{\mathsf{out}} \\ &= ((x-\mathsf{r}_1^{\mathsf{in}}) \mod M)\cdot((y-\mathsf{r}_2^{\mathsf{in}}) \mod N) \\ &\quad + \mathsf{r}^{\mathsf{out}} \mod L\end{aligned}$$

Now, using Equation (2) we have:

$$\begin{aligned}\hat{g}^{[\mathsf{r}_1^{\mathsf{in}},\mathsf{r}_2^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{\mathsf{umult},m,n}(x,y) = (x-\mathsf{r}_1^{\mathsf{in}} + 2^m\cdot\mathbf{1}\{x<\mathsf{r}_1^{\mathsf{in}}\})\cdot(y-\mathsf{r}_2^{\mathsf{in}} \\ + 2^n\cdot\mathbf{1}\{y<\mathsf{r}_2^{\mathsf{in}}\}) + \mathsf{r}^{\mathsf{out}} \mod L\end{aligned}$$

On further expanding we get:

$$\begin{aligned}\hat{g}^{[\mathsf{r}_1^{\mathsf{in}},\mathsf{r}_2^{\mathsf{in}},\mathsf{r}^{\mathsf{out}}]}_{\mathsf{umult},m,n}(x,y) &= 2^n\cdot\mathbf{1}\{y<\mathsf{r}_2^{\mathsf{in}}\}\cdot x \\ &+ 2^m\cdot\mathbf{1}\{x<\mathsf{r}_1^{\mathsf{in}}\}\cdot y \\ &+ 2^n\cdot\mathbf{1}\{y<\mathsf{r}_2^{\mathsf{in}}\}\cdot(-\mathsf{r}_1^{\mathsf{in}}) \\ &+ 2^m\cdot\mathbf{1}\{x<\mathsf{r}_1^{\mathsf{in}}\}\cdot(-\mathsf{r}_2^{\mathsf{in}}) \\ &+ (-\mathsf{r}_1^{\mathsf{in}})\cdot y + (-\mathsf{r}_2^{\mathsf{in}})\cdot x \\ &+ \mathsf{r}_1^{\mathsf{in}}\cdot\mathsf{r}_2^{\mathsf{in}} + \mathsf{r}^{\mathsf{out}} + x\cdot y \mod L\end{aligned}$$

We omit the last term (i.e. $2^m\{x<\mathsf{r}_1^{\mathsf{in}}\}\cdot 2^n\{y<\mathsf{r}_2^{\mathsf{in}}\}$) as it gets cancelled out by the modulus. Based on above rearragment, we present our construction of the FSS gate for unsigned multiplication in Figure 4.

Unsigned Multiplication Gate $(\mathsf{Gen}^{\mathsf{umult}}_{m,n}, \mathsf{Eval}^{\mathsf{umult}}_{m,n})$

$\mathsf{Gen}^{\mathsf{umult}}_{m,n}(1^\lambda, \mathsf{r}_1^{\mathsf{in}}, \mathsf{r}_2^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:

1: Sample random $r_{10}, r_{11} \leftarrow \mathbb{U}_L$ s.t. $r_{10}+r_{11} = \mathsf{uint}_m(-\mathsf{r}_1^{\mathsf{in}}) \mod L$.
2: Sample random $r_{20}, r_{21} \leftarrow \mathbb{U}_L$ s.t. $r_{20}+r_{21} = \mathsf{uint}_n(-\mathsf{r}_2^{\mathsf{in}}) \mod L$.
3: Sample random $r_0, r_1 \leftarrow \mathbb{U}_L$ s.t. $r_0+r_1 = \mathsf{uint}_m(\mathsf{r}_1^{\mathsf{in}})\cdot\mathsf{uint}_n(\mathsf{r}_2^{\mathsf{in}}) + \mathsf{r}^{\mathsf{out}}$.
4: Let $\beta_1 = (1, -\mathsf{r}_2^{\mathsf{in}}) \in \mathbb{U}_N^2$ and $\beta_2 = (1, -\mathsf{r}_1^{\mathsf{in}}) \in \mathbb{U}_M^2$.
5: $(k_{10}, k_{11}) \leftarrow \mathsf{Gen}^{<}_m(1^\lambda, \mathsf{r}_1^{\mathsf{in}}, \beta_1, \mathbb{U}_N^2)$.
6: $(k_{20}, k_{21}) \leftarrow \mathsf{Gen}^{<}_n(1^\lambda, \mathsf{r}_2^{\mathsf{in}}, \beta_2, \mathbb{U}_M^2)$.
7: For $b \in \{0,1\}$, let $k_b = k_{1b}||k_{2b}||r_{1b}||r_{2b}||r_b$.
8: **return** (k_0, k_1).

$\mathsf{Eval}^{\mathsf{umult}}_{m,n}(b, k_b, x, y)$:

1: Parse $k_b = k_{1b}||k_{2b}||r_{1b}||r_{2b}||r_b$.
2: Set $(t_1, t_2) \leftarrow \mathsf{Eval}^{\leq}_m(b, k_{1b}, x)$.
3: Set $(t_3, t_4) \leftarrow \mathsf{Eval}^{\leq}_n(b, k_{2b}, y)$.
4: **return** $x\cdot t_3\cdot 2^n + y\cdot t_1\cdot 2^m + t_4\cdot 2^n + t_2\cdot 2^m + r_{1b}\cdot y + r_{2b}\cdot x + r_b + b\cdot x\cdot y \mod L$.

Fig. 4. FSS Gate for Unsigned Multiplication $\mathcal{G}_{\mathsf{umult}}$, b refers to party id.

Spline Endpoints		Spline coefficients ($a_2x^2 + a_1x + a_0$)		
Left	Right	a_2	a_1	a_0
0	198	-87883	286070	-1
199	398	-74280	148925	169708
399	598	-15013	33009	240687
599	798	-6420	9582	257304
799	32766	0	0	262144
32767	-800	0	0	-262144
-799	-601	6419	-435	-260874
-600	-401	15012	9582	-257305
-400	-201	74279	33009	-240688
-200	-1	87882	148925	-169709

Table 6. Spline intervals (modulo 2^{16}) and coefficients (modulo 2^{32}) for tanh with $n_{\mathcal{I}} = n_{\mathcal{O}} = 16, s_{\mathcal{I}} = s_{\mathcal{O}} = 8, n_c = 32, s_c = 18, d = 2, m = 10$.

B Example of a Spline

In Table 6, we describe the mixed-bitwidth spline for tanh for the following parameters: $n_{\mathcal{I}} = n_{\mathcal{O}} = 16, s_{\mathcal{I}} = s_{\mathcal{O}} = 8, n_c = 32, s_c = 18, d = 2, m = 10$ (see Table 2, Section 5.2). The coefficients a_2, a_1, a_0 (corresponding to decreasing powers of x) for each spline polynomial are fixed-point numbers with bitwidth 32 and scale 18. The endpoints of spline intervals are 16-bit fixed point numbers with scale 8.

C Simple Activations

We now present the construction of FSS protocols for Average Pool (Appendix C.1), as well as ReLU, Maxpool, and Argmax (Appendix C.2).

C.1 Average Pool

The average pool functionality requires signed division of a ring element by a public positive integer. Hence, we start by defining the division functionality, show how to realize it using FSS techniques, and then present our protocol for average pool.

We follow the notation and definitions from [31]. Let $\mathsf{idiv} : \mathbb{Z}\times\mathbb{Z}\to\mathbb{Z}$ denote signed integer division, that is, $\mathsf{idiv}(a,d) = b$ where $a = b\cdot d + r$, $r\in\mathbb{Z}$, $0\leqslant r < d$. Let $\mathsf{rdiv} : \mathbb{S}_N\times\mathbb{Z}\to\mathbb{S}_N$ denote (signed) division of a ring element by a positive integer, defined as

$$\mathsf{rdiv}(a,d) \coloneqq \mathsf{idiv}(\mathsf{sint}(a), d) \mod N$$

where $0 < d < N$. Let $\langle a\rangle_0, \langle a\rangle_1 \in \mathbb{S}_N$ denote additive secret shares of a over $\mathbb{S}_N$, i.e., $\langle a\rangle_0$ and $\langle a\rangle_1$ are random elements of $\mathbb{S}_N$ subject only to the constraint that $(\langle a\rangle_0 + \langle a\rangle_1) \mod N = a$. The following theorem [31] allows expressing $\mathsf{rdiv}(a, d)$ in terms of $\langle a\rangle_0$ and $\langle a\rangle_1$.

Theorem 6 (Division of ring element [31])**.** *Let shares of $a \in \mathbb{S}_N$ be $\langle a\rangle_0, \langle a\rangle_1 \in \mathbb{S}_N$, for some $N = N_1 \cdot d + N_0 \in \mathbb{Z}$, where $N_1, N_0, d \in \mathbb{Z}$ and $0 \leqslant N_0 < d < N$.*

Let the unsigned representation of $a, \langle a\rangle_0, \langle a\rangle_1$ in $\mathbb{S}_N$ lifted to integers be $a_u, a_0, a_1 \in \{0, 1, \ldots, N-1\}$, respectively, such that $a_0 = a_0^1 \cdot d + a_0^0$ and $a_1 = a_1^1 \cdot d + a_1^0$, where $a_0^1, a_0^0, a_1^1, a_1^0 \in \mathbb{Z}$ and $0 \leqslant a_0^0, a_1^0 < d$. Let $N' = \lceil N/2 \rceil$. Let $\mathsf{corr}, A, B, C \in \mathbb{Z}$ be defined as

$$\mathsf{corr} = \begin{cases} -1 & (a_u \geqslant N') \wedge (a_0 < N') \wedge (a_1 < N') \\ 1 & (a_u < N') \wedge (a_0 \geqslant N') \wedge (a_1 \geqslant N') \\ 0 & \textit{otherwise} \end{cases}$$

$$A = a_0^0 + a_1^0 - (\mathbf{1}\{a_0 \geqslant N'\} + \mathbf{1}\{a_1 \geqslant N'\} - \mathsf{corr}) \cdot N_0,$$

$$B = \mathsf{idiv}(a_0^0 - \mathbf{1}\{a_0 \geqslant N'\} \cdot N_0, d) + \mathsf{idiv}(a_1^0 - \mathbf{1}\{a_1 \geqslant N'\} \cdot N_0, d),$$

$$C = \mathbf{1}\{A < d\} + \mathbf{1}\{A < 0\} + \mathbf{1}\{A < -d\}.$$

Then, $\mathsf{rdiv}(a, d) = \mathsf{rdiv}(\langle a\rangle_0, d) + \mathsf{rdiv}(\langle a_1\rangle, d) + (\mathsf{corr} \cdot N_1 + 1 - C - B) \mod N$

In the FSS setting, the dealer holds $\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}} \in \mathbb{S}_N$, while the two parties P_0 and P_1 hold $x \in \mathbb{S}_N$, with the goal of computing $\mathsf{rdiv}(x - \mathsf{r}^{\mathsf{in}}, d) + \mathsf{r}^{\mathsf{out}}$. We will set $\langle a\rangle_0 = x$ and $\langle a\rangle_1 = -\mathsf{r}^{\mathsf{in}}$ in Theorem 6 (i.e. $a = x - \mathsf{r}^{\mathsf{in}} \mod N$).

We will first compute A in the above theorem. To do this, we use the following fact (from [31]). Let $w = \mathbf{1}\{a_0 + a_1 \geqslant N\}$, then $\mathsf{corr} = \mathbf{1}\{a_0 \geqslant N'\} + \mathbf{1}\{a_1 \geqslant N'\} - w - \mathbf{1}\{a \geqslant N'\}$. Now, using $\mathsf{DCF}_{n,\mathbb{S}_N}$, P_0 and P_1 can compute shares of $w = \mathbf{1}\{a_0 + a_1 \geqslant N\} = \mathbf{1}\{N - 1 - a_0 < a_1\}$. Similarly, shares of $\mathbf{1}\{a \geqslant N'\}$ can be computed using the FSS gate for $g_{\mathsf{IC},N',N-1}$. These two computations can be done in parallel in the first round, and from this, the parties can compute shares of $\bar{A} = A + \mathsf{r}^{\mathsf{temp}} \in \mathbb{S}_N$, where $\mathsf{r}^{\mathsf{temp}} \in \mathbb{S}_N$ is a random mask chosen by the dealer.

In the second round, parties first locally compute shares of B. Now, they reconstruct $\bar{A}$, and then, along with an FSS gate for $\mathcal{G}_{\mathsf{sCMP}}$ from [3], compute C. Shares of $\mathsf{rdiv}(x - \mathsf{r}^{\mathsf{in}}, d) + \mathsf{r}^{\mathsf{out}}$ can then be computed locally from shares of B, C and corr. The full FSS protocol for signed division is given in Figure 5.

Theorem 7. *There is a 2-round FSS protocol $(\mathsf{Gen}^{\mathsf{div}}_{n,d}, \mathsf{Eval}^{\mathsf{div}}_{n,d})$ for $\mathcal{G}_{\mathsf{div}}$ which has a total key size of $6n$ bits, plus the key size of $\mathsf{DCF}_{n,\mathbb{S}_N}$, plus the key sizes of FSS gates for $g_{\mathsf{IC},n,\lceil N/2\rceil,N-1}$ and $g_{\mathsf{sCMP},n}$. This protocol requires 1 invocation of $\mathsf{DCF}_{n,\mathbb{S}_N}$, 1 invocation of $\mathsf{Eval}^{\mathsf{IC}}_{n,\lceil N/2\rceil,N-1}$ and 3 invocations of $\mathsf{Eval}^{\mathsf{sCMP}}_n$.*

Remark. In the special case when d is a power of 2, we have $\mathsf{rdiv}(a, d) = (a \gg_A \log_2 d)$, and it is more efficient to use the (single round) arithmetic right-shift (ARS) gate from [3] to perform signed division.

Average pool. The family of functions $\mathcal{G}_{\mathsf{avgpool}}$ to compute the average of d elements is defined as $g_{\mathsf{avgpool},n,d}(x_1, x_2, \ldots, x_d) = (\sum_{i=1}^{d} x_i)/d = \mathsf{rdiv}(\sum_{i=1}^{d} x_i, d) \in \mathbb{S}_N$, where $x_1, x_2, \ldots, x_d \in \mathbb{S}_N$. It is straightforward to derive a 2-round FSS protocol for $\mathcal{G}_{\mathsf{avgpool}}$ from the protocol for signed division.

Theorem 8. *There is a 2-round FSS protocol $(\mathsf{Gen}^{\mathsf{avgpool}}_{n,d}, \mathsf{Eval}^{\mathsf{avgpool}}_{n,d})$ for $\mathcal{G}_{\mathsf{avgpool}}$ which has the same key size and evaluation cost as $(\mathsf{Gen}^{\mathsf{div}}_{n,d}, \mathsf{Eval}^{\mathsf{div}}_{n,d})$.*

C.2 ReLU, Maxpool and Argmax

For the ReLU function, we use the FSS gate for $g_{\mathsf{ReLU},n}$ from the work of [3]. With this gate, one can easily construct an FSS gate to compute the maximum of two elements by defining the function in terms of ReLU – i.e., $g_{\mathsf{max},n}(x_1, x_2) = \mathsf{ReLU}(x_1 - x_2) + x_2$. We then build upon this to construct an FSS protocol for Maxpool (i.e. the function that computes the maximum out of d elements) by computing the maximum of 2 elements at a time in a tree-like manner, resulting in $(d-1)$ comparisons done over $\lceil \log d \rceil$ rounds. Finally, Argmax (that computes the index with the maximum value out of d elements) is computed in a similar manner to Maxpool, in $2\lceil \log d \rceil$ rounds.

Theorem 9. *There is a $\lceil \log d \rceil$-round FSS protocol $(\mathsf{Gen}^{\mathsf{maxpool}}_{n,d}, \mathsf{Eval}^{\mathsf{maxpool}}_{n,d})$ for maxpool on d elements, which has a total key size of $n(d-1)$ bits plus $(d-1)$ times the key size of FSS gate for $g_{\mathsf{ReLU},n}$, and requires $(d-1)$ invocations of $\mathsf{Eval}^{\mathsf{ReLU}}_n$.*

Theorem 10. *There is a $2\lceil \log d \rceil$-round FSS protocol $(\mathsf{Gen}^{\mathsf{argmax}}_{n,d}, \mathsf{Eval}^{\mathsf{argmax}}_{n,d})$ for $\mathcal{G}_{\mathsf{argmax}}$ which has a total key size of $n(d-1)$ bits, plus the key size of FSS protocol for $g_{\mathsf{maxpool},n,d}$, plus $(d-1)$ times the key sizes of FSS protocols for $g_{\mathsf{sCMP},n}$ and $g_{\times,n}$. The protocol requires $(d-1)$ invocations of $\mathsf{Eval}^{\mathsf{ReLU}}_n$, $\mathsf{Eval}^{\mathsf{sCMP}}_n$ and $\mathsf{Eval}^{\times}_n$ each.*

Signed Division $(\mathsf{Gen}^{\mathsf{div}}_{n,d}, \mathsf{Eval}^{\mathsf{div}}_{n,d})$

$\mathsf{Gen}^{\mathsf{div}}_{n,d}(1^\lambda, \mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:

1: Set $\langle a\rangle_1 = (-\mathsf{r}^{\mathsf{in}}) \in \mathbb{S}_N$.
2: Compute $N, N', N_0, N_1, a_1, a_1^1, a_1^0$ as described in Theorem 6.
3: $(k_{10}, k_{11}) \leftarrow \mathsf{Gen}^{<}_n(1^\lambda, a_1, 1, \mathbb{S}_N)$.
4: $(k_{20}, k_{21}) \leftarrow \mathsf{Gen}^{\mathsf{IC}}_{n,N',N-1}(1^\lambda, -a_1, 0)$.
5: Sample random $r_{10}, r_{11} \in \mathbb{S}_N$ s.t. $r_{10} + r_{11} = a_1^0$.
6: Sample random $r_{20}, r_{21} \in \mathbb{S}_N$ s.t. $r_{20} + r_{21} = \mathbf{1}\{a_1 \geqslant N'\}$.
7: Sample random $r_{30}, r_{31} \in \mathbb{S}_N$ s.t. $r_{30} + r_{31} = \mathsf{idiv}(a_1^0 - \mathbf{1}\{a_1 \geqslant N'\} \cdot N_0, d)$.
8: Sample random $r_{40}, r_{41} \in \mathbb{S}_N$ s.t. $r_{40} + r_{41} = \mathsf{rdiv}(\langle a\rangle_1, d)$.
9: Sample random $\mathsf{r}^{\mathsf{temp}} \leftarrow \mathbb{S}_N$ and random $r_{50}, r_{51} \in \mathbb{S}_N$ s.t. $r_{50} + r_{51} = \mathsf{r}^{\mathsf{temp}}$.
10: $(k_{30}, k_{31}) \leftarrow \mathsf{Gen}^{\mathsf{sCMP}}_n(1^\lambda, \mathsf{r}^{\mathsf{temp}}, 0, 0)$
11: Sample random $r_0, r_1 \in \mathbb{S}_N$ s.t. $r_0 + r_1 = \mathsf{r}^{\mathsf{out}}$.
12: For $b \in \{0,1\}$, let $k_b = k_{1b}||k_{2b}||k_{3b}||r_{1b}||r_{2b}||r_{3b}||r_{4b}||r_{5b}||r_b$.
13: **return** (k_0, k_1).

$\mathsf{Eval}^{\mathsf{div}}_{n,d}(b, k_b, x)$:

1: Parse $k_b = k_{1b}||k_{2b}||k_{3b}||r_{1b}||r_{2b}||r_{3b}||r_{4b}||r_{5b}||r_b$.
2: Set $\langle a\rangle_0 = x \in \mathbb{S}_N$.
3: Compute $N, N', N_0, N_1, a_0, a_0^1, a_0^0$ as described in Theorem 6.
4: Set $w_b \leftarrow \mathsf{Eval}^{<}_n(b, k_{1b}, N - 1 - a_0)$.
5: Set $p_b \leftarrow \mathsf{Eval}^{\mathsf{IC}}_{n,N',N-1}(b, k_{2b}, a_0)$.
6: Set $\mathsf{corr}_b = b \cdot \mathbf{1}\{a_0 \geqslant N'\} + r_{2b} - w_b - p_b$.
7: Set $A_b = b \cdot a_0^0 + r_{1b} - (b \cdot \mathbf{1}\{a_0 \geqslant N'\} + r_{2b} - \mathsf{corr}_b) \cdot N_0 + r_{5b}$.
8: Reconstruct $\bar{A} = A_0 + A_1 \in \mathbb{S}_N$.
9: Set $B_b = b \cdot \mathsf{idiv}(a_0^0 - \mathbf{1}\{a_0 \geqslant N'\} \cdot N_0, d) + r_{3b}$.
10: Set $C_{1b} \leftarrow b - \mathsf{Eval}^{\mathsf{sCMP}}_n(b, k_{3b}, \bar{A}, d)$.
11: Set $C_{2b} \leftarrow b - \mathsf{Eval}^{\mathsf{sCMP}}_n(b, k_{3b}, \bar{A}, 0)$.
12: Set $C_{3b} \leftarrow b - \mathsf{Eval}^{\mathsf{sCMP}}_n(b, k_{3b}, \bar{A}, -d)$.
13: Set $C_b = C_{1b} + C_{2b} + C_{3b}$.
14: **return** $b \cdot \mathsf{rdiv}(\langle a\rangle_0, d) + r_{4b} + c_b \cdot N_1 + b - C_b - B_b + r_b \in \mathbb{S}_N$.

Fig. 5. 2-round FSS Protocol for signed division of ring element by a public positive integer.

D Description of Benchmarks

DeepSecure B4. This benchmark from [32] contains 3 fully connected layers, 2 tanh layers with 2000 and 500 instances each and argmax. The uniform bitwidth is set to 16. The input/output scales for tanh are set to 12.

Google-30. This benchmark is an RNN taken from [26] with 99 sigmoid and 99 tanh layers with 100 instances over the Google-30 [40] dataset. The input and output scales for sigmoid are 9 and 14 respectively. The input and output scales for tanh are 9. These two layers operate on bitwidth 16. The network also contains other layers like hadamard product and agrmax.

MiniONN LSTM. This benchmark contains the sigmoid and tanh layers of LSTM from [27] (see Figure 14). It contains a sigmoid layer with 600 instances and a tanh layer with 400 instances. The bitwidth used is 37 and input/output scales are 12 for both layers.

Industrial-72. Since the benchmark is not public, we evaluate the math functions alone for this benchmark as described in [30]. As stated, it contains 7 layers of sigmoid and tanh each with 64 instances in each layer. The bitwidth is used 16 and input/output scales for sigmoid are 8 and 14 respectively. The input/output scales for tanh is set to 8.

Heads. Similar to above, description for this benchmark is not available publicly. This is the only benchmark which contains L2 normalization layers that use reciprocal square root compuation. We use this benchmark to evaluate our protocol for this function. The benchmark contains 3 layers of reciprocal square root each with 1200, 1200 and 300 instances. The first and third layers have input/output scales of 12 and 11. The second layer has input/output scales of 10 and 9. The input/output bitwidths are 16 for all layers.

MiniONN CNN. This is a 7 layer CNN benchmark from [27] (see Figure 13) over CIFAR-10 dataset. This CNN was used as one of the benchmarks in [28]. It contains convolutions, ReLU, and Maxpool layers. The fixed bitwidth and scale of 41 and 15 is used.

ResNet-50. This is a 50 layer CNN from [18] over ImageNet [11] dataset. The code is generated from the publicly available ONNX files. Bitwidth of 37 is used, as done in [31] with a scale of 12.

ResNet-18. This is a 18 layer CNN from [18] over the Hymenoptera dataset. The code is generated from publicly available ONNX files. Bitwidth of 32 is used, as done in [34] with a scale of 10.

E Mixed-bitwidth splines

Figure 6 describes the 3-round FSS protocol for fixed-point mixed-bitwidth spline from Section 5.1 with the text in magenta denoting modifications over the FSS gate for uniform bitwidth splines [3, Figure 6].

Fixed-point mixed-bitwidth spline protocol $(\mathsf{Gen}^{(\text{mixed,fixed})\text{-spline}}_{(n_\mathcal{I},s_\mathcal{I},n_\mathcal{O},s_\mathcal{O},n_c,s_c),m,d,\{p_i\}_i}, \mathsf{Eval}^{(\text{mixed,fixed})\text{-spline}}_{(n_\mathcal{I},s_\mathcal{I},n_\mathcal{O},s_\mathcal{O},n_c,s_c),m,d,\{p_i\}_i})$

$\mathsf{Gen}^{(\text{mixed,fixed})\text{-spline}}_{(n_\mathcal{I},s_\mathcal{I},n_\mathcal{O},s_\mathcal{O},n_c,s_c),m,d,\{p_i\}_i}(1^\lambda, \{f_i\}_i, \mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:

1: Set $N_\mathcal{I} = 2^{n_\mathcal{I}}$, $N_\mathcal{O} = 2^{n_\mathcal{O}}$, $n = n_c + d \cdot n_\mathcal{I}$, $\mathsf{tr} = s_c + d \cdot s_\mathcal{I} - s_\mathcal{O}$.
2: Sample random $\mathsf{r}^{\mathsf{temp}} \leftarrow \mathbb{S}_N$.
3: $(k_{10}, k_{11}) \leftarrow \mathsf{Gen}^{\mathsf{SExt}}_{n_\mathcal{I},n}(1^\lambda, \mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{temp}})$.
4: For $i \in \{1,\ldots,m\}$, let $\bar{f}_i$ be the polynomial corresponding to f_i with coefficients sign extended to n-bits from n_c bits.
5: For $i \in \{1,\ldots,m\}$, let $\beta_i = (f'_{i,d},\ldots,f'_{i,0}) \in \mathbb{S}_N^{(d+1)}$, be the coefficient vector of f'_i s.t. $f'_i(x) = \bar{f}_i(x - \mathsf{r}^{\mathsf{temp}})$.
6: Set $\beta = (\beta_1,\ldots,\beta_m) \in \mathbb{S}_N^{m(d+1)}$ and $\gamma = (N_\mathcal{I} - 1) + \mathsf{r}^{\mathsf{in}} \in \mathbb{S}_{N_\mathcal{I}}$.
7: $(k_0^{(N-1)}, k_1^{(N-1)}) \leftarrow \mathsf{Gen}^{<}_{n_\mathcal{I}}(1^\lambda, \gamma, \beta, \mathbb{S}_N^{m(d+1)})$.
8: **for** $i = \{1,\ldots,m\}$ **do**
9: Set $\alpha_i^{(L)} = p_{i-1} + 1 + \mathsf{r}^{\mathsf{in}} \in \mathbb{S}_{N_\mathcal{I}}$, $\alpha_i^{(R)} = p_i + \mathsf{r}^{\mathsf{in}} \in \mathbb{S}_{N_\mathcal{I}}$ and $\alpha_i^{(R')} = p_i + 1 + \mathsf{r}^{\mathsf{in}} \in \mathbb{S}_{N_\mathcal{I}}$.
10: Set $c_{r,i} = \mathbf{1}\{\alpha_i^{(L)} > \alpha_i^{(R)}\} - \mathbf{1}\{\alpha_i^{(L)} > (p_{i-1} + 1 \bmod N_\mathcal{I})\} + \mathbf{1}\{\alpha_i^{(R')} > (p_i + 1 \bmod N_\mathcal{I})\} + \mathbf{1}\{\alpha_i^{(R)} = N_\mathcal{I} - 1\} \in \mathbb{S}_N$.
11: Sample random $e_{i,0}, e_{i,1} \leftarrow \mathbb{U}_N^{(d+1)}$ s.t. $e_{i,0} + e_{i,1} = c_{r,i} \cdot \beta_i$.
12: Sample random $\beta_{i,0}, \beta_{i,1} \leftarrow \mathbb{U}_N^{(d+1)}$ s.t. $\beta_{i,0} + \beta_{i,1} = \beta_i$.
13: **end for**
14: Sample random $\mathsf{r}^{\mathsf{temp2}} \in \mathbb{S}_N$, and random $r_{10}, r_{11} \leftarrow \mathbb{S}_N$ s.t. $r_{10} + r_{11} = \mathsf{r}^{\mathsf{temp2}}$.
15: **if** $n_\mathcal{O} \leqslant n - \mathsf{tr}$ **then**
16: $(k_{20}, k_{21}) = \mathsf{Gen}^{\mathsf{TR}}_{n,\mathsf{tr}}(1^\lambda, \mathsf{r}^{\mathsf{temp2}}, \mathsf{r}^{\mathsf{out}})$.
17: **else if** $n_\mathcal{O} > n - \mathsf{tr}$ **then**
18: $(k_{20}, k_{21}) = \mathsf{Gen}^{\gg_A}_{n,\mathsf{tr}}(1^\lambda, \mathsf{r}^{\mathsf{temp2}}, \mathsf{r}^{\mathsf{out}})$.
19: **end if**
20: Sample random $r_0, r_1 \leftarrow \mathbb{S}_{N_\mathcal{O}}$ s.t. $r_0 + r_1 = \mathsf{r}^{\mathsf{out}}$.
21: For $b \in \{0,1\}$, let $k_b = k_b^{(N-1)} || \{e_{i,b}\}_i || \{\beta_{i,b}\}_i || r_b || k_{1b} || k_{2b} || r_{1b}$.
22: **return** (k_0, k_1).

$\mathsf{Eval}^{(\text{mixed,fixed})\text{-spline}}_{(n_\mathcal{I},s_\mathcal{I},n_\mathcal{O},s_\mathcal{O},n_c,s_c),m,d,\{p_i\}_i}(b, k_b, x)$:

1: Set $N_\mathcal{I} = 2^{n_\mathcal{I}}$, $N_\mathcal{O} = 2^{n_\mathcal{O}}$, $n = n_c + d \cdot n_\mathcal{I}$, $\mathsf{tr} = s_c + d \cdot s_\mathcal{I} - s_\mathcal{O}$.
2: Parse $k_b = k_b^{(N-1)} || \{e_{i,b}\}_i || \{\beta_{i,b}\}_i || r_b || k_{1b} || k_{2b} || r_{1b}$ and set $\bar{x}_b = \mathsf{Eval}^{\mathsf{SExt}}_{n_\mathcal{I},n}(b, k_{1b}, x)$.
3: Reconstruct $\bar{x} = \bar{x}_0 + \bar{x}_1$.
4: **for** $i = \{1,\ldots,m\}$ **do**
5: Set $x_i = x + (N_\mathcal{I} - 1 - (p_{i-1} + 1)) \in \mathbb{S}_{N_\mathcal{I}}$.
6: Set $(s^{(i)}_{i-1,b}, s^{(i)}_{i,b}) \leftarrow \mathsf{Eval}^{<}_{n_\mathcal{I}}(b, k_b^{(N-1)}, x_i)$.
7: **end for**
8: Set $s^{(m+1)}_{m,b} = s^{(1)}_{m,b}$.
9: **for** $i = \{1,\ldots,m\}$ **do**
10: Set $c_{x,i} = (\mathbf{1}\{x > (p_{i-1} + 1 \bmod N_\mathcal{I})\} - \mathbf{1}\{x > (p_i + 1 \bmod N_\mathcal{I})\}) \in \mathbb{S}_N$.
11: $w_b^{(i)} = (w^{(i)}_{d,b},\ldots,w^{(i)}_{0,b}) = c_{x,i} \cdot \beta_{i,b} - s^{(i)}_{i,b} + s^{(i+1)}_{i,b} + e_{i,b}$.
12: **end for**
13: Set $t_b = (t_{d,b},\ldots,t_{0,b}) = \sum_{i=1}^m w_b^{(i)} \in \mathbb{S}_N^{(d+1)}$. Set $y_b = r_{1b} + \sum_{i=0}^d (t_{i,b} \cdot \bar{x}^i) \in \mathbb{S}_N$.
14: Reconstruct $y = y_0 + y_1 \in \mathbb{S}_N$.
15: **if** $n_\mathcal{O} \leqslant n - \mathsf{tr}$ **then**
16: $z_b = \mathsf{Eval}^{\mathsf{TR}}_{n,tr}(b, k_{2b}, y)$.
17: **else if** $n_\mathcal{O} > n - \mathsf{tr}$ **then**
18: $z_b = \mathsf{Gen}^{\gg_A}_{n,\mathsf{tr}}(b, k_{2b}, y)$.
19: **end if**
20: **return** $z_b \bmod N_\mathcal{O}$.

Fig. 6. FSS protocol for fixed-point mixed-bitwidth spline $g^{(\text{mixed,fixed})}_{\text{spline},(n_\mathcal{I},s_\mathcal{I},n_\mathcal{O},s_\mathcal{O},n_c,s_c),m,d,\{p_i\}_i,\{f_i\}_i}$, b refers to party id.