SIGMA: Secure GPT Inference with Function Secret Sharing

Kanav Gupta*
University of Maryland, College Park
kanav@umd.edu

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

Ananta Mukherjee
Microsoft Research
t-mukherjeea@microsoft.com

Nishanth Chandran
Microsoft Research
nichandr@microsoft.com

Divya Gupta
Microsoft Research
divya.gupta@microsoft.com

Ashish Panwar
Microsoft Research
ashishpanwar@microsoft.com

Rahul Sharma
Microsoft Research
rahsha@microsoft.com

ABSTRACT

Secure 2-party computation (2PC) enables secure inference that offers protection for both proprietary machine learning (ML) models and sensitive inputs to them. However, the existing secure inference solutions suffer from high latency and communication overheads, particularly for transformers. Function secret sharing (FSS) is a recent paradigm for obtaining efficient 2PC protocols with a preprocessing phase. We provide SIGMA, the first end-to-end system for secure transformer inference based on FSS. By constructing new FSS-based protocols for complex machine learning functionalities, such as Softmax, GeLU and SiLU, and also accelerating their computation on GPUs, SIGMA improves the latency of secure inference of transformers by $11-19\times$ over the state-of-the-art that uses preprocessing and GPUs. We present the first secure inference of generative pre-trained transformer (GPT) models. In particular, SIGMA executes Meta's Llama2 (available on HuggingFace) with 13 billion parameters in 44 seconds and GPT2 in 1.6 seconds.

KEYWORDS

Function Secret Sharing, MPC, secure inference, transformers, GPT

1 INTRODUCTION

In the problem of secure inference, model providers own proprietary machine learning (ML) models that they want to offer as services and clients who want to learn the inference results on their sensitive data. The security requirement is that the client should learn nothing about the model beyond the inference output and the model provider should learn nothing about the client's input. This problem can be solved by the technique of secure 2-party computation (2PC) that provides cryptographic security guarantees.

In recent years, the applicability of 2PC-based solutions has scaled up from models with thousands of parameters [12, 42, 52, 54–56, 59, 63, 65, 67, 78], to models with millions of parameters [23, 35, 38, 41, 46, 64, 79, 82], to BERT models with hundreds of millions of parameters [9, 22, 36, 45, 49]. In this paper, we take a step further in this direction by providing secure inference of Generative Pre-trained Transformer (GPT) models with billions of parameters.

Transformer-based generative language models have gained significant traction in recent times due to their remarkable performance on various natural language tasks e.g., question-answering, summarization, language translation, code generation [17, 18, 72]. Apart from ensuring model/input privacy, secure inference of such models opens up other interesting scenarios like "prompt privacy". AI companies are spending significant efforts building prompts that lead to good inference results and they want to keep the prompts hidden. Secure inference allows a company holding a proprietary prompt and a client holding sensitive data to generate inference results from a public language model without revealing their inputs to each other. However, the current state-of-the-art systems for secure inference deliver unsatisfactory results on transformers.

We posit that a system for secure ML inference must satisfy the following requirements: (1) *accuracy* - i.e., the accuracy under secure inference should match that of the plaintext, (2) *security* - i.e., the system should provide standard 2PC security, (3) *efficiency* - i.e., the latency and communication overheads of secure inference should be low, and (4) *scalability* - i.e., the system must scale to models with billions of parameters. We show that existing systems fail to meet (often more than one of) these requirements.

Existing secure transformer inference systems include THE-X [22], Iron [36], and CrypTen [45, 49, 81] (we discuss other works in Section 8). THE-X sacrifices both accuracy, by replacing *complex* non-linearities (based on elementary functions, e.g., e^x) with simple non-linearities ($\max(x, 0)$), and security, by revealing intermediate values. Iron maintains both accuracy and security, but has huge communication overheads, requiring over a hundred GB of communication even for BERT models. Although CrypTen leverages GPU acceleration and preprocessing to improve efficiency, its online latency and communication for secure inference are still significant. Moreover, it fails to provide standard 2PC security because it uses insecure[1] *local* truncations. Furthermore, because of GPU memory overflows, it fails to scale to larger models.

1.1 Our Contributions

In this paper, we propose SIGMA[2] - a system that advances the state-of-the-art for secure inference of transformer-based models along multiple dimensions. Like CrypTen, SIGMA works in 2PC with preprocessing model and uses GPU acceleration, but is an *order of magnitude* more efficient in latency and communication while providing standard 2PC security guarantees. SIGMA maintains the

*Equal Contribution. Work done while at Microsoft Research.

[1] Secure inference works like CrypTen [45] and many others [56, 70, 78, 79, 82] use cheap local truncations that have recently been established as insecure [50].

[2] Secure Inference of GPT Models Accelerated

model accuracy under secure inference through precise approximations of complex non-linearities and scales efficiently to GPT models with billions of parameters.

Sigma leverages Function Secret Sharing (FSS) based 2PC protocols [13, 16, 35, 41] and builds on Orca [41] that is the state-of-the-art in GPU-accelerated FSS-protocols. However, Orca focuses primarily on convolutional neural networks (CNNs) that use simple non-linearities like ReLU. We show that Orca's techniques pose unacceptable overheads for transformers because of their heavy use of complex non-linearities (Section 7.1.2).

Since the latency of secure inference in transformers is dominated by complex non-linearities - GeLU, SiLU, Softmax, layer normalization [36] - we propose new FSS-based protocols for these operations and accelerate them with GPUs. Realizing these operations requires accurate computation of various elementary functions, e.g., exponentiation, reciprocal square root, inverse, etc. The prior work of Pika [77] uses large look-up tables (LUTs) for these functions. Although this approach is general, Grotto [69] shows that large LUTs are inefficient and provides protocols based on custom splines (when they exist). Sigma's protocols minimize the size of LUTs, to maintain accuracy, while being more efficient than Grotto (Section 7.1.1). For instance, for GeLU over 50-bit values, while Pika requires an LUT of size 2^{50}, Sigma uses an LUT of size 2^8 and overall requires 9× lower compute than Grotto in the same threat model.

We evaluate Sigma on models based on GPT [17], BERT [26] and Llama2 [73], which are widely used for next-word-predictions and classification tasks. Our novel protocols securely and accurately evaluate GPT-Neo with 1.3 billion parameters – "a transformer model designed using EleutherAI's replication of the GPT-3 architecture" [3] – in 7.4 seconds. Sigma also supports the Llama2 models recently released by Meta AI and available on Huggingface. It takes 27 seconds for Llama2-7B [5] and 44 seconds for Llama2 13B [4]. Sigma runs the smaller GPT2 model [2] from HuggingFace (tens of millions of downloads each month) in 1.6 seconds, and the BERT models in $0.1 - 4.7$ seconds. Overall, Sigma improves the latency of secure inference by $11.5 - 19.4\times$ over the state-of-the-art.

To guarantee standard 2PC security, Sigma does away with local truncations and instead uses secure faithful truncations. Truncations are used extensively in both linear layers, i.e., after matrix multiplications, and non-linear layers. We provide a new protocol for faithful truncation (Section 4.2) that is much more efficient than the prior work [13] (up to 30×). Even though our truncations are costlier than (almost free) local truncations in CrypTen, our massive performance gains in GeLU, SiLU and Softmax make Sigma more than 10× faster than CrypTen for end-to-end inference.

Our large scale evaluations are made possible by Sigma's frontend that allows users to succinctly express a transformer architecture of choice and run it with Sigma's protocols optimized for CPUs or GPUs (Section 6). The protocol design for CPUs and GPUs differ, and we support both (Section 5.1). In fact, Sigma running on CPUs is already faster than CrypTen running on GPUs. We discuss some real world considerations when using SIGMA in Appendix A. Sigma will be made publicly available.

2 PRELIMINARIES

2.1 Notation

Let λ be the computational security parameter, $N = 2^n$ and $L = 2^\ell$. Let $\mathbb{R}$ denote the set of real numbers and $\mathbb{U}_{2^n}$ denote the set of n-bit unsigned integers. We use standard 2's complement representation to represent signed values in $\mathbb{U}_N$. For $x \in \mathbb{U}_N$, $\mathsf{int}_n(x)$ and $\mathsf{uint}_n(x)$ denote the corresponding signed and unsigned integers in $\mathbb{Z}$, respectively. We denote arrays using boldface and its i-th element (starting at 0) using the same symbol in normal typeface followed by $[i]$, e.g., $\boldsymbol{a} = \{a[0], a[1], a[2], \dots\}$.

2.1.1 Fixed-Point Representation. Fixed-point representation, parameterized by bitwidth n and precision f, encodes a real value $r \in \mathbb{R}$ into an n-bit integer $x \in \mathbb{U}_N$ such that $x = \left\lfloor r \cdot 2^f \right\rfloor \mod N$. Conversely, an n-bit fixed-point number x with precision f decodes into real number $\frac{\mathsf{int}_n(x)}{2^f}$.

2.1.2 Operators. For a predicate b, $1\{b\} \in \{0, 1\}$ returns 1 if b is true and 0 otherwise. For $n < \ell$, $x \in \mathbb{U}_N$, $\mathsf{extend}_{n,\ell}(x)$ returns x appended with $(\ell - n)$ 0's on the left. For $x \in \mathbb{U}_N$, $\mathsf{MSB}_n(x) \in \{0, 1\}$ denotes the most-significant bit of x.

2.1.3 Secret Sharing. For $x \in \mathbb{U}_N$, *secret sharing* samples random *shares* $x_0, x_1 \in \mathbb{U}_N$ such that $x = x_0 + x_1 \mod N$ holds, and is denoted by $\mathsf{share}(x)$. When x_0 is held by P_0 and x_1 is held by P_1, we denote the process of exchanging the shares and adding them to reconstruct the underlying value by $\mathsf{reconstruct}(x_b)$ for $b \in \{0, 1\}$.

2.2 Threat Model

This work considers standard 2PC in the preprocessing model [10, 11, 16, 25, 40] that has also received significant attention in the context of secure machine learning [35, 41, 45, 68, 69, 82]. That is, there are two parties P_0 and P_1 with inputs x_0 and x_1 and they wish to compute a public function $y = f(x_0, x_1)$ without revealing anything more than the function output y to each other. In a preprocessing phase that is independent of the inputs to the function x_0 and x_1, correlated randomness is generated and made available to P_0 and P_1. This randomness can be generated in multiple ways: a trusted dealer [13, 16, 35, 41, 45, 68, 69, 82], generic 2PC protocols [31, 84], or through specialized 2PC protocols [27]. In this work, we consider the first approach. All our protocols satisfy the standard notion of simulation-based security [19, 31, 51] with security provided against a semi-honest static probabilistic polynomial time (PPT) adversary corrupting either P_0 or P_1.

2.3 Function Secret Sharing

A Function Secret Sharing (FSS) [14, 15] scheme is a pair of algorithms (Gen, Eval). Gen splits a function g into two function shares (g_0, g_1) and Eval takes as input $b \in \{0, 1\}$, function share g_b and input x and returns $g_b(x)$. The correctness property of an FSS scheme requires that $g_0(x) + g_1(x) = g(x)$ for all x. The security property requires that each function share g_b hides the function g.

Definition 1 (FSS: Syntax [14, 15]). *A (2-party) FSS scheme is a pair of algorithms* (Gen, Eval) *such that:*

- $\mathsf{Gen}(1^\lambda, \hat{g})$ *is a PPT key generation algorithm that given* 1^λ *and* $\hat{g} \in \{0, 1\}^*$ *(description of a function g) outputs a pair of keys*

(k_0, k_1). *We assume that $\hat{g}$ explicitly contains descriptions of input and output groups $\mathbb{G}^{\text{in}}, \mathbb{G}^{\text{out}}$.*

- $\text{Eval}(b, k_b, x)$ *is a polynomial-time evaluation algorithm that given $b \in \{0,1\}$ (party index), k_b (key defining $g_b : \mathbb{G}^{\text{in}} \to \mathbb{G}^{\text{out}}$) and $x \in \mathbb{G}^{\text{in}}$ (input for g_b) outputs $y_b \in \mathbb{G}^{\text{out}}$ (the value of $g_b(x)$).*

(k_0, k_1) are called FSS keys and the number of bits required to store one FSS key is called *key size*. We formally define correctness and security of an FSS scheme in Appendix B.

2.4 2PC with preprocessing from FSS

Consider secure computation of a circuit with gates $\{g_i\}_i$ and wires $\{w_i\}_i$. We describe the 2PC protocol with preprocessing using FSS from [16] in two phases.

2.4.1 Offline Phase. For each wire w_i, sample a random mask r_i from the appropriate group. Then, for each of the gate g with input wire w_i and output wire w_j, generate an FSS key for its offset function $g^{[r_i, r_j]}(x) = g(x - r_i) + r_j$ and provide one key to each party. For input and output wires of the circuit belonging to party b, that party also learns the masks associated with those wires.

2.4.2 Online Phase. For each input wire w_i with value x_i owned by a party b, party b calculates $\hat{x}_i = x_i + r_i$ and sends it to party $1-b$. Now, the parties evaluate the circuit gates in topological order. To evaluate a gate g with input and output wire w_i and w_j respectively, both parties evaluate the corresponding FSS key on $\hat{x}_i$ to get secret shares of $\hat{x}_j = g^{[r_i, r_j]}(\hat{x}_i) = g(\hat{x}_i - r_i) + r_j = g(x_i) + r_j$. The parties then reconstruct these shares to get masked value $\hat{x}_j$. For the output wires, the party owning the wire subtracts the corresponding mask to get the final output value.

2.4.3 Protocol Structure and Security for FSS protocols. We use $\hat{(\cdot)}$ to denote masked values. Consider a function F and input x such that $y = F(x)$. Protocol for F, denoted by Π^F, has two phases Gen^F and Eval^F. Gen^F is executed in the preprocessing phase on input and output masks r^{in} and r^{out}, respectively, to produce the preprocessing material or *keys* for F made available to P_0 and P_1. The number of bits required to store the key for Π^F is called the *key size* and is denoted by $\text{keysize}(\Pi^F)$. Next, Eval^F is the protocol run by P_0 and P_1 in the online phase on masked input $\hat{x} = x + \mathsf{r}^{\text{in}}$ and their respective keys. At the end of Eval^F, P_0 and P_1 learn secret-shares of masked output value $\hat{y} = y + \mathsf{r}^{\text{out}}$. All protocols presented in this paper have the above structure.

Security for $\Pi^F = (\text{Gen}^F, \text{Eval}^F)$ is defined through the following two interactions. 1) A *real interaction* in which Gen^F is executed in the preprocessing phase (with input and output masks r^{in} and r^{out}) and P_0 and P_1 execute Eval^F in the online phase with keys obtained in the preprocessing phase. This interaction happens in the presence of an adversary $\mathcal{A}$ and the environment $\mathcal{Z}$. 2) An *ideal interaction* in which P_0 and P_1 send their inputs to a functionality that computes the functionality faithfully (i.e., unmasks $\hat{x}$ to get x, computes $y = F(x)$, computes $\hat{y} = y + \mathsf{r}^{\text{out}}$ and provides shares of $\hat{y}$ to P_0 and P_1). We say that protocol Π^F securely realizes function F if for every adversary $\mathcal{A}$ in the real interaction, there is an adversary $\mathcal{S}$ (called the simulator) in the ideal interaction such that no environment $\mathcal{Z}$ can distinguish between the two interactions.

2.5 Distributed Point Function (DPF)

The point function $f^{\bullet}_{\alpha,\beta} : \mathbb{U}_N \to \mathbb{G}^{\text{out}}$ takes as input $x \in \mathbb{U}_N$ and outputs $\beta \in \mathbb{G}^{\text{out}}$ if $x = \alpha$ and 0 otherwise. The corresponding FSS-scheme for point function $(\text{Gen}^{\bullet}_n, \text{Eval}^{\bullet}_n)$ is called *Distributed Point Function* [14, 15]. Notationally, we write $(k^{\bullet}_0, k^{\bullet}_1) \leftarrow \text{Gen}^{\bullet}_n(1^\lambda, \alpha, \beta, \mathbb{G}^{\text{out}})$ and $y_b = \text{Eval}^{\bullet}_n(b, k^{\bullet}_b, x)$, for $x \in \mathbb{U}_N$. For all our protocols, it suffices to have $\mathbb{G}^{\text{out}} = \{0,1\}$ and $\beta = 1$, and this allows us to leverage the construction of DPF with *early termination optimization* (that is applicable for small payloads) [15].

Theorem 1 (Cost of DPF from [15]). *Given PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$ and let $\nu = \log_2(\lambda + 1)$. When $n > \nu$, there exists a DPF for $f^{\bullet}_{\alpha,1} : \mathbb{U}_N \to \{0,1\}$ with key size $(n-\nu)\cdot(\lambda+2) + 2\lambda$. Number of PRG invocations in $\text{Gen}^{\bullet}_n$ is $2(n-\nu)$ and in $\text{Eval}^{\bullet}_n$ is $n - \nu$. When $n \leqslant \nu$, keysize of 2^n and 0 PRG invocations in $\text{Gen}^{\bullet}_n$ and $\text{Eval}^{\bullet}_n$ is required.*

Similar to prior FSS works [15, 69, 77], we set $\lambda = 127$ and implement the required length doubling PRG using 2 calls to AES-128 in counter mode. As previously observed [15, 69], a single AES call suffices for $\text{Eval}^{\bullet}_n$ as only half of the output is used. From here on, we refer to it as an *half-PRG call.*

2.6 Comparisons using DPF Keys

Comparison function $f^{<}_{\alpha,\beta} : \mathbb{U}_N \to \mathbb{G}^{\text{out}}$ takes as input $x \in \mathbb{U}_N$ and returns $\beta \in \mathbb{G}^{\text{out}}$ if $x < \alpha$ and 0 otherwise. Previous works [13, 35] used a specialized FSS-scheme called *Distributed Comparison Function* (DCF) to realize this functionality. Recent work of [69] showed that when $\mathbb{G}^{\text{out}} = \{0,1\}$, $\beta = 1$, FSS scheme for comparison function can be constructed using the DPF construction from [15].

Theorem 2 (FSS for comparison using DPF [69]). *There exists an algorithm $\text{Eval}^{<}_n$ such that $\forall x, \alpha \in \mathbb{U}_N$:*

$$(k^{\bullet}_0, k^{\bullet}_1) \leftarrow \text{Gen}^{\bullet}_n(1^\lambda, \alpha, 1, \{0,1\})$$
$$\implies \text{Eval}^{<}_n(0, x, k^{\bullet}_0) + \text{Eval}^{<}_n(1, x, k^{\bullet}_1) = f^{<}_{\alpha,1}(x)$$

and $\text{Eval}^{<}_n$ invokes DPF half-PRG $\max(n-\nu, 0)$ times. Thus, $(\text{Gen}^{\bullet}_n, \text{Eval}^{<}_n)$ is an FSS-scheme for comparison function.

Compared to DCF construction from [13] that requires a length quadrupling PRG, the above construction lowers compute by $> 2\times$.

3 OVERVIEW OF TRANSFORMERS

3.1 Architecture Overview

Transformers is a neural network architecture used commonly in natural language tasks. At a high level, a transformer architecture consists of an encoder and a decoder [76]. The encoder generates a sequence of hidden states from the given input sequence. The decoder takes the hidden states produced by the encoder and generates the output sequence. Real-world models stack multiple encoder and decoder blocks, as shown in Figure 1, to obtain high accuracy results. Further, transformers can be used in both encoder-decoder (e.g., BERT) and decoder-only mode (e.g., GPT). We discuss the key components of a single transformer block below:

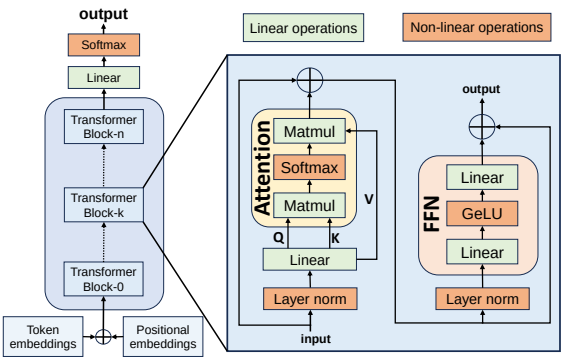

Figure 1: Architecture of a transformer neural network

3.1.1 Token embeddings. Transformers represent a natural language input as a sequence of tokens (e.g., each word can be represented as a token) wherein each token is a one-dimensional vector of size d_{model}. The token embedding matrix $W_e \in \mathbb{R}^{d_{model} \times N_V}$, where N_V is the vocabulary size, maps each token to its corresponding embedding vector. Further, each token is also assigned a positional encoding vector of size d_{model} that encodes the token's position in the input sequence [76]. The sum of the token embedding vector and the positional encoding vector is used as input to the model.

3.1.2 Self-attention and multi-head attention (MHA). The self attention mechanism helps the model attend to different parts in the input sequence. It maps a query and a set of key-value pairs to an output as follows:

$$\text{Attention (Q, K, V)} = \text{softmax}(\text{QK}^T/\sqrt{d_{model}})\text{V}$$

where $Q \in \mathbb{R}^{y \times d_{model}}$ is the query matrix and $K, V \in \mathbb{R}^{z \times d_{model}}$ are key and value matrices, respectively (here, y and z represent the length of primary and context sequence.)

The multi-head attention module consists of multiple attention heads that operate in parallel, each over $\frac{d_{model}}{num\_heads}$ in the above formulation (e.g., $num\_heads = 12$ in GPT-2). The outputs of the attention heads are concatenated and linearly transformed to obtain the MHA output.

3.1.3 Softmax: For a vector $\boldsymbol{x} \in \mathbb{R}^k$, define $x_{\max} = \max(x_0, x_1, \ldots, x_{k-1})$. The softmax function on $\boldsymbol{x}$ returns a vector $\boldsymbol{y} \in \mathbb{R}^k$ s.t.:

$$y[i] = \frac{e^{x[i]}}{\sum_{j=0}^{k-1} e^{x[j]}} = \frac{e^{x[i]-x_{\max}}}{\sum_{j=0}^{k-1} e^{x[j]-x_{\max}}}$$

Since exponentials in the first expression can get arbitrarily large, the second expression is preferred where exponential is only computed on negative values (including 0).

3.1.4 Feed forward network (FFN). FFN consists of two fully connected layers wherein the first layer transforms the input from dimension d_{model} to d_{ff}, and the second layer transforms it back to d_{model} (typically, $d_{ff} = 4 \times d_{model}$). FFN for a matrix $X \in \mathbb{R}^{z \times d_{model}}$ (where z is the sequence length) can be represented as:

$$\text{FFN}(X) = \text{GeLU}(XW_1 + b_1)W_2 + b_2$$

where $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$ are the weight matrices and $b_1 \in \mathbb{R}^{d_{ff}}$, $b_2 \in \mathbb{R}^{d_{model}}$ are the bias vectors for first and second layers within FFN. GeLU is the Gaussian Error Linear Unit activation function [37]. The Llama2 models [73] use SiLU, a special variant of the Swish activation function [61], instead of GeLU.

3.1.5 Activation. An activation function applies a non-linear transformation element-wise to the given input vector and its output determines which of the neurons should be activated in the next layer. Popular examples of activation functions include ReLU, GeLU, tanh etc. Most of our models use GeLU, which returns a vector $\boldsymbol{y} \in \mathbb{R}^k$ for $\boldsymbol{x} \in \mathbb{R}^k$ s.t.:

$$y_i = \text{GeLU}(x_i) = \frac{x_i}{2}(1 + \text{erf}\,(\frac{x_i}{\sqrt{2}}))$$

where erf is an integral of a Gaussian [37]. The Llama2 models use SiLU, which returns

$$y_i = \text{SiLU}(x_i) = x_i \,.\, \sigma(x_i) = \frac{x_i}{1+e^{-x_i}}$$

where $\sigma(x)$ is the Sigmoid function.

3.1.6 Layer normalization. Layer norm normalizes the distribution of activations at each layer in the model. For a vector of real values $\boldsymbol{x} \in \mathbb{R}^k$, let $m = \sum x_i/k$ and $v = (\sum (x_i - m)^2)/k$ denote its mean and variance, respectively. For $z_i = x_i - m$ and model parameters $\gamma, \beta \in \mathbb{R}$, layer normalization returns a vector $\boldsymbol{y} \in \mathbb{R}^k$ s.t.:

$$y_i = \gamma \cdot \frac{x_i - m}{\sqrt{v}} + \beta = \gamma \cdot \frac{z_i}{\sqrt{\sum z_i^2/k}} + \beta \tag{1}$$

Root Mean Squared Normalization (RMS Norm) is another kind of normalization (used in the Llama2 models [73]) and is computationally simpler and more efficient than Layer Norm. For model parameter $\gamma \in \mathbb{R}$, the RMS Norm of $\boldsymbol{x} \in \mathbb{R}^k$ is a vector $\boldsymbol{y} \in \mathbb{R}^k$ s.t.:

$$y_i = \gamma \cdot \frac{x_i}{RMS(\boldsymbol{x})} = \gamma \cdot \frac{x_i}{\sqrt{\sum x_i^2/k}} \tag{2}$$

3.2 Secure Inference of Transformers

Based on the above description and the literature on cryptographic protocols, the layers in a transformer can be classified into two categories - linear and non-linear.

3.2.1 Linear layers. These consist of the matrix multiplications occurring in multihead attention (MHA) and feed forward (FFN) layers. Similar to all prior works on secure inference, we work with fixed-point arithmetic. Here, multiplying two fixed-point values with precision f over integers results in a fixed-point value with implicit precision $2f$. Hence, multiplications must be followed by a truncation operation to bring the precision back to f. For the matrix multiplications over integers, we use the existing protocol [16, 35, 41] that relies on Beaver-triple like correlations generated in preprocessing phase. For truncations, as one of our contributions, we provide a significantly more efficient protocol compared to the prior work [13, 35] (see Section 4.2).

3.2.2 Non-linear layers. These consist of GeLU, SiLU, Softmax, LayerNorm and RMSNorm. In Section 5, we provide novel precise protocols for these non-linearities over fixed-point arithmetic that not only preserve the accuracy of transformers but also lead to efficient secure inference on transformers (Section 7).

3.2.3 Putting things together. For each of the layers of the transformers, we provide a secure protocol where the evaluating parties start with masked input, i.e., for an input x and random mask r^{in}, parties hold $\hat{x} = x + \mathsf{r}^{\mathsf{in}}$ and after the protocol learn masked output, i.e., $\hat{y} = y + \mathsf{r}^{\mathsf{out}}$ for output y and mask $\mathsf{r}^{\mathsf{out}}$. Given this invariant, we are able to trivially put together the secure protocols for each layer to obtain a secure protocol for inference and prove security by invoking the sequential composition theorem [19, 51].

3.3 GPU-accelerated Secure Inference

Graphics Processing Units (GPUs) support thousands of concurrent threads and provide much higher memory bandwidth compared to CPUs [6]. Therefore, GPUs are a natural fit for accelerating transformers in plaintext: (1) several linear layers (e.g., in FFN) in a transformer network involve large matrix multiplications that can be accelerated using GPUs, often by up to two orders of magnitude compared to CPUs. (2) the non-linear layers are memory intensive and hence benefit from the high memory bandwidth of GPUs. Under secure inference, the linear layers can be accelerated similar to plaintext. However, the non-linear layers require several rounds of network communication between the client and the model provider, and transfer of large pre-generated keys from CPU to GPU over the PCIe links. Therefore, communication and key transfer overheads dominate the overall time under secure inference.

We reduce the size of communication and data transfer, at the expense of some extra computation, as follows: (1) we reduce network communication with an efficient packing scheme for non-standard bitwidths. This adds extra computation for packing and unpacking values which we implement efficiently on the GPU itself. (2) we reduce the number of DPF keys needed for GeLU/SiLU from two to one at the cost of one extra evaluation of the same key per element. These optimizations reduce network communication by $1.2 - 1.5\times$ and key transfer by $1.8\times$ over a naïve port of our CPU protocols to the GPU. Note that without these optimizations, a GPU's compute units would often remain idle. Hence, the additional computation is essentially free for Sigma.

4 CRYPTO BUILDING BLOCKS

Similar to Orca [41], we design efficient protocols with multi-round online phase. Our goal is to achieve low key size, online compute and online communication while ensuring small constant round complexity. At the end of Eval^F, the evaluators learn secret-shares of masked output value $\hat{y} = y + \mathsf{r}^{\mathsf{out}}$. Now, Eval^F can be followed by a reconstruct to obtain the masked output value $\hat{y}$ and we denote this modified protocol by $\hat{\Pi}^F$. As the input and output masks are unknown to the evaluators, the cleartext values remain hidden from the evaluators.

We first provide a summary of protocols for multiplication, selection, and lookup tables from prior works. Then, we describe our novel FSS-based protocols for truncation and comparison. All of these are used as sub-protocols by our novel protocols for complex non-linearities (Section 5).

4.1 Protocols from Previous Works

4.1.1 Multiplication. For secure multiplication of two n-bit integers, [16] provides a beaver-triple based (non-interactive) FSS-protocol Π_n^{Mul} with keysize of $3n$ bits.

4.1.2 Select. The functionality $\mathsf{select}_n : \{0,1\} \times \mathbb{U}_N \to \mathbb{U}_N$ takes as input a selector bit s and a payload x such that $\mathsf{select}_n(s, x) = x$ if $s = 1$ and 0 otherwise. Orca [41] provides a non-interactive protocol Π_n^{select} that realizes select_n securely with keysize $4n$.

4.1.3 SelectLin. Let $\mathsf{selectlin}_{n,\gamma} : \{0,1\}^2 \times \mathbb{U}_N \to \mathbb{U}_N$ be a functionality parameterized by a length 4 vector of pairs of elements, $\gamma = \{(\alpha_0, \beta_0), (\alpha_1, \beta_1), (\alpha_2, \beta_2), (\alpha_3, \beta_3)\}$ with $\alpha_i, \beta_i \in \mathbb{U}_N, \forall i \in [4]$. It takes as input two selector bits s_0, s_1, and a payload x, and outputs $\mathsf{selectlin}_{n,\gamma}(s_0, s_1, x) = \alpha_{2s_0+s_1} x + \beta_{2s_0+s_1}$. This functionality can be easily realized using one-time truth tables as described in [24] and results in a non-interactive protocol $\Pi_n^{\mathsf{selectlin}_\gamma}$ with keysize $8n$.

4.1.4 Look-up Table. The functionality $\mathsf{LUT}_{n,\ell,T} : \mathbb{U}_N \to \mathbb{U}_L$ is parameterized by input bitwidth n, output bitwidth ℓ and a public table $\boldsymbol{T} \in \mathbb{U}_L^N$. It takes as input $x \in \mathbb{U}_N$ and returns $T[x] \in \mathbb{U}_L$. Pika [77] provides a protocol $\Pi_{n,\ell,\boldsymbol{T}}^{\mathsf{LUT}}$ such that $\mathsf{keysize}(\Pi_{n,\ell,\boldsymbol{T}}^{\mathsf{LUT}}) = \mathsf{keysize}(\mathsf{DPF}_{n,1}) + n + 2\ell$. Online phase invokes the DPF PRG $2^{n-\nu} - 1$ times, where $\nu = \log_2(\lambda + 1)$, and communicates 2ℓ bits in 1 round.

4.2 Our Truncation Protocol

As discussed in Section 3.2, linear layers or matrix multiplication needs to be followed by an element-wise truncation to bring down precision. Our protocols for complex non-linearities also use multiple truncations. The literature considers (cheap) local truncations [45, 56, 78, 79, 82] and (expensive) faithful truncations [13, 35, 64]. While local truncations are almost free to implement, a very recent work Li et al. [50] shows that these do not satisfy standard simulation-based security and are insecure. In light of this, in this work, all our protocols for secure inference only use faithful truncations or arithmetic right shifts (ARS). Here, we provide new protocols for truncation that are much more efficient than prior FSS-based protocol from [13, 35].

4.2.1 ARS with guaranteed gap. We first consider the case when the input is known have a *gap* w.r.t. the bitwidth used. In particular, we require that $v \in \mathbb{U}_N$ is such that $v \in [0, 2^{n-2}) \cup [2^n - 2^{n-2}, 2^n)$. Looking ahead, within our protocols for non-linearities, this assumption holds many a times from domain knowledge.

We first use the following relation from [28] to reduce ARS to logical right shift (LRS), i.e., a reduction of shift of signed values to unsigned values. In particular, for n-bit values and shift amount f, when $v \in [0, 2^{n-2}) \cup [2^n - 2^{n-2}, 2^n)$, for $x = v + 2^{n-2}$,

$$\mathsf{ARS}_{n,f}(v) = \mathsf{LRS}_{n,f}(x) - 2^{n-f-2}$$

where $\mathsf{LRS}_{n,f}(x) = \left\lfloor \frac{x}{2^f} \right\rfloor$. Note that constraint on v implies that $x = v + 2^{n-2}$ seen as an unsigned value lies in $[0, 2^{n-1})$ which would be crucial for the optimization that we do.

Now, given the above relation, to construct a protocol for $\mathsf{ARS}_{n,f}(v)$, we construct a protocol for $\mathsf{LRS}_{n,f}(x)$ using the following lemma (also used in [13, 64]).

Logical Right-Shift with Gap $\Pi^{\mathsf{GapLRS}}_{n,f}$

$\mathsf{Gen}^{\mathsf{GapLRS}}_{n,f}(\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:

1: $(k^{\bullet}_0, k^{\bullet}_1) \leftarrow \mathsf{Gen}^{\bullet}_f(1^{\lambda}, \mathsf{r}^{\mathsf{in}} \mod 2^f, 1, \{0,1\})$
2: $c \xleftarrow{\$} \{0,1\}, r^{(w)} = \mathsf{extend}_{1,n}(c)$
3: $m = 2^{n-f} \cdot \mathsf{extend}_{1,n}(\mathsf{MSB}_n(\mathsf{r}^{\mathsf{in}}))$
4: $r = \mathsf{r}^{\mathsf{out}} - \mathsf{LRS}_{n,f}(\mathsf{r}^{\mathsf{in}})$
5: share $(r^{(w)}, m, r)$
6: For $b \in \{0,1\}$, $k_b = k^{\bullet}_b || r^{(w)}_b || m_b || r_b$

$\mathsf{Eval}^{\mathsf{GapLRS}}_{n,f}(b, k_b, \hat{x})$:

1: Parse k_b as $k^{\bullet}_b || r^{(w)}_b || m_b || r_b$
2: $\hat{w}_b = \mathsf{Eval}^{<}_f(b, k^{\bullet}_b, \hat{x} \mod 2^f) + r^{(w)}_b \mod 2$
3: $\hat{w} = \mathsf{reconstruct}(\hat{w}_b)$, $\hat{z} = \mathsf{extend}_{1,n}(\hat{w})$
4: $u_b = b\hat{z} + r^{(w)}_b - 2\hat{z} r^{(w)}_b$
5: $t_b = m_b \cdot \mathsf{extend}_{1,n}(1 - \mathsf{MSB}_n(\hat{x}))$
6: **return** $b \cdot \mathsf{LRS}_{n,f}(\hat{x}) + r_b + t_b - u_b$

Figure 2: Protocol for Logical Right-Shift with Gap

Lemma 1. *For* $x_0 = \hat{x} \mod 2^f$ *and* $r_0 = \mathsf{r}^{in} \mod 2^f$,

$$\mathsf{LRS}^{[\mathsf{r}^{in}, \mathsf{r}^{out}]}_{n,f}(\hat{x}) = \mathsf{LRS}_{n,f}(\hat{x}) - \mathsf{LRS}_{n,f}(\mathsf{r}^{in}) + 2^{n-f} \cdot 1\{\hat{x} < \mathsf{r}^{in}\} - 1\{x_0 < r_0\} + \mathsf{r}^{out} \quad (3)$$

When $x \in [0, 2^{n-1})$, following observation[3] (proof in Appendix C) provides an efficient way to compute $1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\}$.

Lemma 2. *For* $\hat{x} = x + \mathsf{r}^{in} \mod N$, *if* $x < 2^{n-1}$,

$$1\{\hat{x} < \mathsf{r}^{in}\} = 1\{\mathsf{MSB}_n(\hat{x}) = 0\} \wedge 1\{\mathsf{MSB}_n(\mathsf{r}^{in}) = 1\}$$

We provide a formal description of our protocol for LRS for inputs with a gap in Figure 2 (security proof in Appendix D.1). Here, the term $1\{x_0 < r_0\}$ is computed using DPF-based comparison with 1-bit output to allow for smaller FSS key and lower online compute. Once the evaluators learn the masked value of this bit ($\hat{w}$), they do a local extension ($\hat{z}$). They use $\hat{z}$ and arithmetic shares of the mask ($r^{(w)}$) provided by the dealer to obtain arithmetic shares of $u = 1\{x_0 < r_0\}$. It is trivial to extend this to ARS (with the same cost) and we summarize the cost in Theorem 3.

Theorem 3. *There exists a protocol* $\Pi^{\mathsf{GapARS}}_{n,f}$ *that realizes* $\mathsf{ARS}_{n,f}$ *securely for cleartext inputs in* $[0, 2^{n-2}) \cup [2^n - 2^{n-2}, 2^n)$ *such that* $\mathsf{keysize}(\Pi^{\mathsf{GapARS}}_{n,f}) = \mathsf{keysize}(\mathsf{DPF}_{f,1}) + 3n$. *The online phase requires* 1 *evaluation of* $\mathsf{DPF}_{f,1}$ *and communication of* 2 *bits in 1 round.*

4.2.2 Truncate-Reduce. $\mathsf{TR}_{n,f} : \mathbb{U}_N \to \mathbb{U}_{2^{n-f}}$ is defined as dropping the lower f bits of the n-bit input and returning the output as an $(n-f)$-bit number. It can be expressed as:

$$\mathsf{TR}_{n,f}(x) = \mathsf{LRS}_{n,f}(x) \mod 2^{n-f}$$

[3]Similar observation was also used by [28] for their probabilistic LRS protocol that ignores the LSB correction term $1\{x_0 < r_0\}$ and referred to as MSB-to-Wrap optimization in Sirnn [63] and used in various protocols.

Note that Equation 3 for LRS does not rely on gap in inputs. Now, as the term $2^{n-f} \cdot 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\}$ cancels out due to mod operation, we can realize truncate-reduce securely using a single comparison for $1\{x_0 < r_0\}$. We omit details and summarize cost below:

Theorem 4. *There exists a protocol* $\Pi^{\mathsf{TR}}_{n,f}$ *that realizes* $\mathsf{TR}_{n,f}$ *securely such that* $\mathsf{keysize}(\Pi^{\mathsf{TR}}_{n,f}) = \mathsf{keysize}(\mathsf{DPF}_{f,1}) + 2(n-f)$. *The online phase requires* 1 *evaluation of* $\mathsf{DPF}_{f,1}$ *and communicates* 2 *bits in 1 round.*

4.2.3 ARS without known gap. Let $\mathsf{SignExt}_{\ell,n} : \mathbb{U}_L \to \mathbb{U}_N$ be defined as sign extending a value in ℓ-bits to equivalent value in n-bits. When input to ARS is not known to have a gap, we express[4] $\mathsf{ARS}_{n,f}$ as $\mathsf{TR}_{n,f}$ followed by $\mathsf{SignExt}_{n-f,n}$. We use our protocol for (faithful) truncate-reduce and replace DCF in the protocol for sign-extension from Orca [41] with DPF-based comparison. We summarize overall costs below:

Theorem 5. *There exists a protocol* $\Pi^{\mathsf{ARS}}_{n,f}$ *that realizes* $\mathsf{ARS}_{n,f}$ *securely such that* $\mathsf{keysize}(\Pi^{\mathsf{ARS}}_{n,f}) = \mathsf{keysize}(\Pi^{\mathsf{TR}}_{n,f}) + \mathsf{keysize}(\mathsf{DPF}_{n-f,1}) + 2n + 1$. *Online phase requires* 1 *evaluation each of* $\mathsf{DPF}_{f,1}$ *and* $\mathsf{DPF}_{n-f,1}$ *and communicates* $2(n-f)+4$ *bits in 3 rounds.*

4.2.4 Cost Comparison. In contrast, [13] gave a protocol for $\mathsf{ARS}_{n,f}$ (also used in [35]) that requires a key size of approximately $n(\lambda + 2n) + f(\lambda + n)$ bits and online phase makes $2(n + f - 1)$ AES calls. Concretely, for $n = 64$, $f = 12$, $\Pi^{\mathsf{GapARS}}_{n,f}$ has 17.5× smaller key size and 30× lower online compute, and $\Pi^{\mathsf{ARS}}_{n,f}$ has 2.5× smaller key size and 3× lower online compute.

4.3 Our DReLU and Comparison Protocols

For an n-bit value $x \in \mathbb{U}_N$ in 2's complement notation, derivative of ReLU or DReLU is defined as

$$\mathsf{DReLU}_n(x) = 1\{x < 2^{n-1}\} = 1 \oplus \mathsf{MSB}_n(x)$$

and the offset function of DReLU_n can be written as

$$\mathsf{DReLU}^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}_n(\hat{x}) = \mathsf{DReLU}_n(\hat{x} - \mathsf{r}^{\mathsf{in}} \mod N) \oplus \mathsf{r}^{\mathsf{out}}$$
$$= \mathsf{MSB}_n(\hat{x} - \mathsf{r}^{\mathsf{in}} \mod N) \oplus 1 \oplus \mathsf{r}^{\mathsf{out}}$$

Prior FSS works [13, 16, 35, 41] provide a non-interactive protocol for DReLU that uses a DCF key for comparison and evaluates it twice during online phase. In contrast, we provide a non-interactive protocol that does a single evaluation of a DPF key for comparison. In all, we get > 4× reduction in online compute.

Our protocol builds on the logic used in CrypTFlow2 [64] for MSB computation over secret shares (in $\log n$ rounds). For $x \in \mathbb{U}_N$ such that $x = x_0 + x_1 \mod N$, $y_0 = x_0 \mod 2^{n-1}$ and $y_1 = x_1 \mod 2^{n-1}$,

$$\mathsf{MSB}_n(x) = \mathsf{MSB}_n(x_0) \oplus \mathsf{MSB}_n(x_1) \oplus 1\{y_0 + y_1 \geq 2^{n-1}\}$$

Using this above, we get

$$\mathsf{DReLU}^{[\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}}]}_n(\hat{x}) = \mathsf{MSB}_n(\hat{x}) \oplus \mathsf{MSB}_n(2^n - \mathsf{r}^{\mathsf{in}}) \oplus 1\{2^{n-1} - y_0 - 1 < y_1\} \oplus 1 \oplus \mathsf{r}^{\mathsf{out}}$$

where $y_0 = \hat{x} \mod 2^{n-1}$ and $y_1 = (2^n - \mathsf{r}^{\mathsf{in}}) \mod 2^{n-1}$.

[4]Similar approach was used in Orca [41] for stochastic truncations.

DReLU Π_n^{DReLU}

$\mathsf{Gen}_n^{\mathsf{DReLU}}(\mathsf{r}^{\mathsf{in}}, \mathsf{r}^{\mathsf{out}})$:

1: $x_1 = 2^n - \mathsf{r}^{\mathsf{in}}$
2: $y_1 = x_1 \mod 2^{n-1}$
3: $(k_0^\bullet, k_1^\bullet) \leftarrow \mathsf{Gen}_{n-1}^\bullet(1^\lambda, y_1, 1, \{0,1\})$
4: $r = \mathsf{r}^{\mathsf{out}} \oplus \mathsf{MSB}_n(x_1) \oplus 1$
5: share r
6: For $b \in \{0,1\}$, $k_b = r_b || k_b^\bullet$

$\mathsf{Eval}_n^{\mathsf{DReLU}}(b, k_b, \hat{x})$:

1: Parse k_b as $r_b || k_b^\bullet$
2: $\hat{y} = \hat{x} \mod 2^{n-1}$
3: $t_b \leftarrow \mathsf{Eval}_{n-1}^{<}(b, k_b^\bullet, 2^{n-1} - \hat{y} - 1)$
4: **return** $b \cdot \mathsf{MSB}_n(\hat{x}) \oplus r_b \oplus t_b$

Figure 3: Protocol for DReLU.

Based on above equation, we provide a protocol for DReLU_n in Figure 3 (security proof in Appendix D.1) where we compute $1\{2^{n-1} - y_0 - 1 < y_1\}$ using a single DPF-based comparison.

Theorem 6. Π_n^{DReLU} *(non-interactively) securely realizes* DReLU_n *with* $\mathsf{keysize}(\Pi_n^{\mathsf{DReLU}}) = \mathsf{keysize}(\mathsf{DPF}_{n-1,1}) + 1$. *Online phase requires* 1 *evaluation of* $\mathsf{DPF}_{n-1,1}$.

Comparison. To compare two values x, y, i.e., to compute $x \geqslant y$, similar to all prior works, we re-write it as $x - y \geqslant 0$ and realize it using a call to Π_n^{DReLU}.

5 OUR PROTOCOLS FOR COMPLEX NON-LINEARITIES

Here, we describe our protocols for various complex non-linearities - GeLU and SiLU (Section 5.1), softmax (Section 5.2), and layer normalization (Section 5.3). Finally, in Section 5.4, we discuss a few transformers-specific optimizations that allow us to compute these non-linearities over smaller tensors or smaller bitwidths in certain scenarios. Computing these non-linear functions requires efficient computation of various unary functions - GeLU, SiLU, exponential, inverse, and reciprocal square root. Pika's approach to compute any arbitrary elementary function is to just look up the correct output from a table [77]. However, for an n-bit input, it requires a lookup table (LUT) of size 2^n, and computing it securely requires roughly 2^{n-7} PRG calls. In contrast, Grotto [69] uses custom splines and DPFs to realize a subset of functions required in transformers (see Section 7 for a thorough comparison).

In Sigma, we devise function-dependent strategies to significantly reduce the size of LUTs used, while ensuring that our protocols provide good numerical approximations and hence, preserve the accuracy of transformers when run securely using our protocols. For $f = 12$ used by all our benchmarks, our protocols use LUTs of size 2^8 for GeLU and exponential, 2^{10} for SiLU, an LUT of size between 2^{13} and 2^{16} for inverse, and an LUT of size 2^{13} for reciprocal square root, independent of bitwidth n. Note that almost all our benchmarks require a bitwidth of around 50 and our techniques result in significantly smaller LUTs than Pika that are very efficient to compute securely. Moreover, our recipe for approximating reciprocal square root is general and applicable to any elementary function.

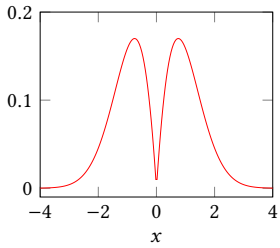

Figure 4: Plot for $\delta(x) = \mathsf{ReLU}(x) - \mathsf{GeLU}(x)$.

$\mathsf{GeLU}_{n,f}(x)$:

1: $p = \mathsf{ReLU}_n(x)$
2: $c = \mathsf{Clip}_{n,A,B}(x)$
3: $a = \mathsf{Abs}_n(c)$
4: $t = \mathsf{TR}_{n,f-6}(a) \mod 256$
5: **return** $p - \mathsf{LUT}_{8,n,T}(t)$

Figure 5: Our approximation for $\mathsf{GeLU}_{n,f}(x)$.

For each of the non-linearities, we describe our secure protocol as a sequence of calls to protocols described in Section 4 and security trivially holds in the simulation paradigm using sequential composition [19, 51]. While for ease of exposition, we describe our ideas for $f = 12$ that is used by all our transformer benchmarks, they can easily be generalized to higher precision values by using appropriately larger LUTs.

5.1 GeLU

For a real number x, $\mathsf{GeLU}(x) = 0.5x(1 + \mathsf{erf}(x/\sqrt{2}))$ where erf is the error function [37]. Prior works, e.g., Crypten [45], Grotto [69], provide protocols for GeLU in the same threat model as ours. However, these are an order of magnitude less performant than Sigma (Section 7.1).

Our main insight is that $\mathsf{GeLU}(x)$ is same as $\mathsf{ReLU}(x) := \max(x, 0)$ almost everywhere except in a small interval around 0. Let $\delta(x) = \mathsf{ReLU}(x) - \mathsf{GeLU}(x)$ (plot shown in Figure 4). Given that $\mathsf{ReLU}(x)$ can be efficiently realized using a call to DReLU and select, it suffices to efficiently compute $\delta(x)$ for x near 0. Finally, we output $\mathsf{GeLU}(x)$ as $\mathsf{ReLU}(x) - \delta(x)$. We calculate $\delta(x)$ using an LUT. However, for efficiency, we need to restrict the input domain of the LUT, while ensuring that the results are precise enough.

First, we observe that $\delta(x)$ becomes negligible outside the range $(-4, 4)$ and for precision $f = 12$, $\delta(-4) = \delta(4) = 0$. Hence, we first restrict the inputs to $(-4, 4)$ or equivalently $[-2^{f+2} + 1, 2^{f+2} - 1]$ using a *clip* operation. Formally, for n-bit values and clipping nodes A, B, we define $\mathsf{Clip}_{n,A,B}(x)$ as (i) A for $x < A$ (ii) x for $x \in [A, B]$, and (iii) B for $x > B$.

Next, we observe that $\delta(x)$ is an even function between $(-4, 4)$. Hence, it suffices to compute the LUT using the absolute value of the clipped input, that lies in $[0, 2^{f+2} - 1]$ and requires $f + 2$ bits to represent. We further reduce the size of input domain to LUT by scaling down to 6-bits of precision, retaining 8-bits of information that are used as input to the LUT to compute $\delta(x)$.

We provide a formal description of our approximation of $\mathsf{GeLU}(x)$ in Figure 5. Here, $A = -2^{f+2} + 1$ and $B = 2^{f+2} - 1$. Also, $T \in \mathbb{U}_N^{256}$ is the table such that for all $i \in \mathbb{U}_{256}$, $T[i] = \left\lfloor \delta(\frac{i}{2^6}) \cdot 2^f \right\rfloor$. For $f = 12$, our approximation achieves[5] an ULP error of 31 which

[5]We compute error by exhaustive testing on all inputs between $(-4, 4)$ as the error is 0 outside this domain.

GeLU (CPU) $\Pi^{\mathsf{GeLUCPU}}_{n,m,f}(\hat{x})$

1: $\hat{y} \leftarrow \hat{\Pi}^{\mathsf{TR}}_{m,f-6}(\hat{x} \mod 2^m)$
2: $\hat{d} \leftarrow \hat{\Pi}^{\mathsf{DReLU}}_{m-f+6}(\hat{y})$
3: $\hat{p} \leftarrow \hat{\Pi}^{\mathsf{select}}_{m-f+6}(\hat{d}, \hat{y})$
4: $\hat{a} \leftarrow 2 \cdot \hat{p} - \hat{y}$
5: $\hat{i} \leftarrow \hat{\Pi}^{\mathsf{DReLU}}_{m-f+6}(\hat{a} - 256) \oplus 1$
6: $\hat{c} \leftarrow \hat{\Pi}^{\mathsf{select}}_{8}(\hat{i}, \hat{a} - 255 \mod 256) + 255$
7: **return** $\Pi^{\mathsf{select}}_{n}(\hat{d}, \hat{x}) - \Pi^{\mathsf{LUT}}_{8,n,T}(\hat{c})$

Figure 6: CPU-optimized protocol for $\mathsf{GeLU}_{n,m,f}$

GeLU (GPU) $\Pi^{\mathsf{GeLUGPU}}_{n,m,f}(\hat{x})$

1: $\hat{y} \leftarrow \hat{\Pi}^{\mathsf{TR}}_{m,f-6}(\hat{x} \mod 2^m)$
2: $\hat{d}_b \leftarrow \Pi^{\mathsf{DReLU}}_{m-f+6}(\hat{y})$
3: $\hat{i}_b \leftarrow \Pi^{\mathsf{DReLU}}_{m-f+6}(\hat{y} + 255) \oplus \Pi^{\mathsf{DReLU}}_{m-f+6}(\hat{y} - 256)$
4: $(\hat{i}, \hat{d}) = \mathsf{reconstruct}(\hat{i}_b, \hat{d}_b)$
5: $\hat{z} = \hat{y} \mod 256$
6: $\hat{c} \leftarrow \hat{\Pi}^{\mathsf{selectlin}_\gamma}_{8}(\hat{i}, \hat{d}, \hat{z})$
7: **return** $\Pi^{\mathsf{select}}_{n}(\hat{d}, \hat{x}) - \Pi^{\mathsf{LUT}}_{8,n,T}(\hat{c})$

Figure 7: GPU-optimized protocol for $\mathsf{GeLU}_{n,m,f}$**. The calls to** Π^{DReLU} **in steps 2-3 can use same key.**

suffices to maintain PyTorch accuracy for all benchmarks as shown in Section 7.

Next, we describe how we translate the above cleartext function to secure protocols. We do a re-ordering of operations in the above description to achieve secure operations on lower bitwidths, resulting in lower keysize, online compute, and communication. Moreover, since the performance bottlenecks are different on CPU and GPU, we provide two different versions of the GeLU protocol. Looking ahead, for GPUs, we trade-off lower keysize and communication with higher compute compared to CPU.

5.1.1 CPU Protocol. We make the following optimizations.

Optimization 1. Since $A = -B$, it holds that $\mathsf{Abs}_n(\mathsf{Clip}_{n,A,B}(x)) = \mathsf{Clip}_{n,0,B}(\mathsf{Abs}_n(x))$. Hence, we switch the steps (2) and (3) in Figure 5 to $a = \mathsf{Abs}_n(x)$; $c = \mathsf{Clip}_{n,0,B}(a)$. This switch has 2 benefits. First, the absolute value can be calculated for free given ReLU as $\mathsf{Abs}_n(x) = 2 \cdot \mathsf{ReLU}_n(x) - x$. Second, since the input to Clip is now guaranteed to be a positive number, it can be realized by 1 comparison (with B) instead of 2 before (one each with A and B).

Optimization 2. Since the lower $f-6$ bits are going to be discarded anyways, and do not affect the outcome of comparisons in ReLU or Clip, it is safe to perform this operation as the very first step. This reduces the bitwidth of comparisons in ReLU and Clip by $f-6$.

Optimization 3. This applies when domain knowledge helps in restricting the inputs of GeLU to a sub-domain of $\mathbb{U}_N$. For instance, in all transformers, GeLU is always preceded by a linear layer that invokes a truncation by f after a matrix multiplication. Due to this, the effective input bitwidth of the GeLU input is $m = n - f$. Combining this with the above, the comparisons can happen over $m - (f-6)$ bits.

Based on the above optimizations, we present our CPU-optimized protocol $\Pi^{\mathsf{GeLUCPU}}_{n,m,f}$ for $\mathsf{GeLU}_{n,m,f}$ in Figure 6, where input/output bitwidths are n, effective input bitwidth is m, and precision is f.

Cost Analysis. $\Pi^{\mathsf{GeLUCPU}}_{n,m,f}$ requires a key size equal to the key size of 2 $\Pi^{\mathsf{DReLU}}_{m-f+6}$, 1 $\Pi^{\mathsf{LUT}}_{8,n,T}$, 1 $\Pi^{\mathsf{TR}}_{m,f-6}$ and 3 calls to Π^{select} of bitwidths n, $m-f+6$ and 8. Online phase compute consists of a single evaluation of these and communication of $4(m-f)+2n+46$ bits in 6 rounds.

5.1.2 GPU Protocol. We note that the performance bottlenecks on CPU and GPU are quite different. CPU implementations are bottlenecked by compute (i.e., number of AES calls). However, once AES calls are accelerated well on GPU, performance bottlenecks become key transfer from CPU RAM to GPU memory and communication between the two parties. Thus, when creating a secure version of Figure 5 for the GPU, we focus on reducing key size and communication while tolerating a higher compute. We later argue that this trade-off results in lower runtime compared to a naïve port of the CPU protocol.

Our starting point is the protocol outlined in Figure 6. To allow computing on smaller bitwidths, we keep optimizations 2 and 3 intact. Thus, we start by computing $y = \mathsf{TR}_{m,f-6}(x \mod 2^m)$. Crucially, we let go of optimization 1, and combine ReLU and Clip differently. First, we compute DReLU bit $d = \mathsf{DReLU}(y)$. We additionally compute an interval containment bit $i = 1\{-255 \leqslant y \leqslant 255\} = \mathsf{DReLU}(y-256) - \mathsf{DReLU}(y+255)$. In doing so, we compute one more DReLU than the CPU, i.e., a total of 3. However, crucially, since all the DReLU evaluations are on y shifted by a constant, they can all use the same key. Hence, unlike GeLUCPU, this requires a *single* DPF key.

Given i and d, we compute $\mathsf{Abs}(\mathsf{Clip}(y))$ as 255 when $i = 0$, and when $i = 1$, as $-y$ when $d = 0$ and y when $d = 1$. As an optimization, similar to CPU, before a selection, we first reduce y to 8 (relevant) bits and compute $\mathsf{Abs}(\mathsf{Clip}(z))$, where $z = y \mod 256$. Note that since i is computed on y, it only allows the value of z to propagate when $-255 \leqslant y \leqslant 255$. Since d already contains the sign of y, the last 8 bits of y (captured by z), suffice to correctly compute $\mathsf{Abs}(\mathsf{Clip}(y))$. For this selection based on i and d, we invoke $\hat{\Pi}^{\mathsf{selectlin}_\gamma}_{8}(i, d, z)$ with $\gamma = \{(0, 255), (0, 255), (-1, 0), (1, 0)\}$. This gives us $c = \mathsf{Abs}(\mathsf{Clip}(z))$.

We provide the formal GPU protocol in Figure 7. We also note that unlike the CPU version, this does not require reconstructing $\mathsf{ReLU}(x)$ over $m-f-6$ bits (Step 3 in Figure 6). This is because we extract the interval containment bit needed for Clip from x and not from $\mathsf{Abs}(x)$. This allows us to save on communication as well as one round, resulting in efficient GPU implementation.

Cost Analysis. $\Pi^{\mathsf{GeLUGPU}}_{n,m,f}$ requires a key size equal to the key size of 1 DPF_{m-f+5} key (for the 3 calls to $\hat{\Pi}^{\mathsf{DReLU}}_{m-f+6}$), 1 $\Pi^{\mathsf{LUT}}_{8,n,T}$ call, 1 $\hat{\Pi}^{\mathsf{TR}}_{m,f-6}$ call and 2 calls to Π^{select} for bitwidths 8 and n. The online phase communicates $2(m-f)+2n+34$ bits in 4 rounds.

Compared to CPU protocol for $n = 64, m = 52, f = 12$, the GPU protocol has 1.8× smaller keysize, 1.3× less communication, and 1.5× larger number of half-PRG calls. Empirically, on a microbenchmark of 1 million GeLUs, our protocol takes about 70ms, of which 34ms is key transfer, 16ms is compute (of which about 88% is DReLU) and 20ms is communication. This is about 1.4× faster than a naïve port of the CPU protocol.

Our CPU and GPU protocols for SiLU are similar and outlined in Appendix E. Our techniques can be extended to support other activations as well, and some of these are described in Appendix F.

5.2 Softmax

For a vector $\boldsymbol{x} \in \mathbb{R}^k$ and $x_{\max} = \max(x_0, x_1, \ldots, x_{k-1})$, softmax on $\boldsymbol{x}$ returns a vector $\boldsymbol{y} \in \mathbb{R}^k$ such that:

$$y[i] = \frac{e^{x[i]-x_{\max}}}{\sum_{j=0}^{k-1} e^{x[j]-x_{\max}}}$$

5.2.1 Overview. We need protocols for max, exponentiation of negative values and inverse. $x_{\max}$ can be computed using $k-1$ invocations of our protocols for comparison of 2 elements (Section 4.3) and select in $2\lceil \log_2(k) \rceil$ rounds. Now, we can subtract $x_{\max}$ from every element $x[i]$ to obtain $x[i] - x_{\max}$ and invoke the exponentiation protocol on this value to obtain $z[i]$. We can then compute $z = \sum_{j=0}^{k-1} z[i]$, invoke our protocol for inverse on z to obtain z^{-1}, and compute $y[i] = z^{-1} \cdot z[i]$ followed by truncation. We use $\Pi_{n,f}^{\mathsf{GapARS}}$ for the final truncation as $y[i] \in [0,1]$ with precision $2f$ (due to being a probability distribution) resulting in the required *gap*.

Below, we describe novel and efficient protocols for exponential and inverse that we built using domain knowledge of softmax.

5.2.2 Negative Exponential. Define $\mathsf{nExp}(x) = e^{-x}$ for $x \in \mathbb{R}^+$. We observe that nExp is a monotonically decreasing function and for $f = 12$, $x \geqslant 16$, fixed-point representation of $\mathsf{nExp}(x)$, i.e., $\lfloor e^{-x} \cdot 2^{12} \rfloor = 0$. Hence, we first clip the inputs to the interval $[0, 16) \subset \mathbb{R}^+$ followed by using an LUT to compute nExp for this interval. When $x \in [0, 16)$, we need 16-bits to represent fixed-point values with precision $f = 12$. Now, directly using lookup for exponentiation would require an expensive LUT of size 2^{16}.

Next, we use the technique from Seedot [32] for nExp (also used in [63]) that allows reducing one 16-bit LUT to two 8-bit LUTs. Let $c = c_1 || c_0$ be the 16-bit clipped value with $f = 12$, where c_1 is upper 8-bits and c_0 is lower 8-bits. These can be calculated as $c_1 = \mathsf{TR}_{16,8}(c)$ and $c_0 = c \mod 256$. Seedot showed that

$$\left\lfloor \mathsf{nExp}\left(\frac{c}{2^{12}}\right) \cdot 2^f \right\rfloor \approx \mathsf{ARS}_{n,f}\left(\boldsymbol{T_1}[c_1] \cdot \boldsymbol{T_0}[c_0]\right)$$

where $\boldsymbol{T_1}, \boldsymbol{T_0}$ are 8-bit LUTs with n-bit values such that $T_1[i] = \lfloor \mathsf{nExp}(i/2^4) \cdot 2^f \rfloor$ and $T_0[i] = \lfloor \mathsf{nExp}(i/2^{12}) \cdot 2^f \rfloor$ for all $i \in \mathbb{U}_{2^8}$. Here, $\Pi_{n,f}^{\mathsf{GapARS}}$ suffices to perform $\mathsf{ARS}_{n,f}$ as its input is always less than 2^{2f}, leading to a gap. Compared to using 16-bit LUT, the above approach reduces online compute by 100× (1022 half-PRG calls to 10 half-PRG calls including TR and ARS).

We provide a formal description of our protocol $\Pi_{n,m,f}^{\mathsf{nExp}}$ in Figure 8. Here, similar to GeLU, we introduce an additional parameter m that captures effective bitwidth and helps reduce cost when possible from domain knowledge.

Negative Exponential $\Pi_{n,m,f}^{\mathsf{nExp}}(\hat{x})$

1: $\hat{d} \leftarrow \hat{\Pi}_m^{\mathsf{DReLU}}((\hat{x} - 2^{16}) \mod 2^m) \oplus 1$
2: $\hat{c} \leftarrow \hat{\Pi}_{16}^{\mathsf{select}}(\hat{d}, \hat{x} - (2^{16} - 1) \mod 2^{16}) + (2^{16} - 1)$
3: $\hat{c}_1 \leftarrow \hat{\Pi}_{16,8}^{\mathsf{TR}}(\hat{c})$; $\hat{c}_0 \leftarrow \hat{c} \mod 256$
4: $\hat{t}_1 \leftarrow \hat{\Pi}_{8,n,\boldsymbol{T_1}}^{\mathsf{LUT}}(\hat{c}_1)$; $\hat{t}_0 \leftarrow \hat{\Pi}_{8,n,\boldsymbol{T_0}}^{\mathsf{LUT}}(\hat{c}_0)$
5: $\hat{t} \leftarrow \hat{\Pi}_n^{\mathsf{Mul}}(\hat{t}_0, \hat{t}_1)$
6: **return** $\Pi_{n,f}^{\mathsf{GapARS}}(\hat{t})$

Figure 8: Protocol for $\mathsf{nExp}_{n,m,f}$

5.2.3 Inverse. We calculate inverse using an LUT of carefully chosen size. It is easy to see that for a softmax of size k, the input to inverse $z \in [1, k]$. That is, it has a non-zero integer part which is also upper bounded. Hence, without losing any information, we reduce the bitwidth of input from n to $p = f + \lceil \log_2(k+1) \rceil$ retaining precision f. Next, we create an approximate input with lower precision by chopping off few lower bits[6]. In our specific case, we reduce precision to 6, creating an input with bitwdth $q = 6 + \lceil \log_2(k+1) \rceil$. Finally, we use a q-bit LUT to read the output of inverse. The protocol for inverse $\Pi_{n,f}^{\mathsf{Inv}}$ returns $\Pi_{q,n,\boldsymbol{T}}^{\mathsf{LUT}}\left(\hat{\Pi}_{p,f-6}^{\mathsf{TR}}(\hat{x} \mod 2^p)\right)$, where $\boldsymbol{T} \in \mathbb{U}_N^{2^q}$ is a table such that $T[i] = \left\lfloor 2^{f+6}/i \right\rfloor$ for all $i \in \mathbb{U}_{2^q}$.

5.3 Layer Normalization

Equation 1, Section 3.1 provides the mathematical expression for layer normalization. We note that all sub-expressions in the equation can be implemented using our existing protocols barring reciprocal square root. Below we provide an overview of our protocol for reciprocal square root and defer the details of the overall protocol and an additional optimization to Appendix G. The same techniques yield a protocol for RMS Norm (Equation 2, Section 3.1) as well.

5.3.1 Reciprocal Square Root. While we aim to approximate the reciprocal square root using an LUT, securely computing an n-bit LUT for a large n (e.g., 50) is not efficient. So far, we have exploited two main ideas to reduce the size of LUTs significantly. Either, the function is non-zero only in a small domain (e.g., $\mathsf{GeLU}(x) - \mathsf{ReLU}(x)$, $\mathsf{nExp}(x)$) or we use domain knowledge to restrict the input domain (e.g., inverse in softmax). However, both these ideas are inapplicable here. Although reciprocal square root is a monotonically decreasing function, it only approximates to 0 for very large values. Moreover, we do not have any useful lower or upper bound on the input. Hence, our idea is to shift to a representation that allows representing a large dynamic range with a small number of bits. This is exactly what floating-point representations allow. We use domain knowledge to design a custom 13-bit floating-point representation to encode the input and use it to index an LUT. We provide a formal description in Appendix G.2.

[6]While doing this in general can lose all information from the input and result in garbage result for inverse, it is still safe to do in our setting because the initial input has a meaningful lower bound.

5.4 Global Optimizations

5.4.1 Effective Bitwidth. In transformers, GeLU/SiLU is always preceded by a linear layer which invokes a truncation after matrix multiplication. This means, for n bit inputs to the linear layer, the output of truncation by f lies in range $[-2^{n-f-1}, 2^{n-f-1})$. Hence, the effective bitwidth of the input to GeLU is only $m = n - f$. This lets us perform comparisons on a smaller bitwidth m instead of n.

Similarly, softmax is also preceded by a linear layer. As the first step of softmax is to find the max element, all the comparisons in max calculation can happen over an effective bitwidth of $m = n - f$. Then, the max element is subtracted from all the elements in the input vector before being passed to the protocol for nExp. As both input vector elements and max element have effective bitwidth of $n - f$, the input to nExp has effective bitwidth of $m = n - f + 1$.

5.4.2 Attention Mask. In transformer models, for input with sequence length k, the input to softmax is always a batch of k vectors of size k. In many GPT models, including those that we evaluate on, the upper triangular elements of the softmax input are *masked*, i.e., their nExp is set to 0 in the softmax computations. Hence, we can avoid calling the max and nExp protocols for the masked elements and reduce their number of calls to half.

6 IMPLEMENTATION

We have implemented two versions of Sigma, one which is optimized for CPUs and the other for GPUs.

6.0.1 GPU. The GPU code has around 9K lines of C++ and CUDA code. For the GPU version, our starting point is Orca [41], which is currently the state-of-the-art in GPU-accelerated FSS. Similar to [41, 82], we use CUTLASS [1] to implement linear layers. We borrow Orca's ideas on AES acceleration, memory layout and payload packing to build an efficient GPU-accelerated DPF kernel. Securely realizing $\mathsf{LUT}_{n,\ell,T}$ (Section 4.1) requires computing $\mathsf{Eval}^{\bullet}_n(b, k^{\bullet}_b, x),\ \forall x \in \mathbb{U}_N$ [77]. For this, we follow the depth-first approach of [47], while using Orca's AES kernel.

Building on our optimized kernels for DPFs and LUTs, we provide efficient GPU implementations of our protocols for GeLU, SiLU, Softmax, LayerNorm and RMSNorm. We carefully use templating as in Orca [41] and Piranha [82] to ensure that compute happens on lower bitwidths wherever possible. In GeLU, for example, we use the fact that selectIn (Step 6 in Figure 7) runs on $z \in \mathbb{U}_{256}$ to run the protocol with the `uint8_t` data-type on the GPU. This helps us reduce key size, which, in turn, reduces the time to transfer keys from CPU to GPU memory.

Once the compute has been accelerated, key transfer and communication dominate most of the runtime. For example, communication and key transfer consume 35% and 44% of the total runtime. To lower communication, we observe that our protocols operate on non-powers-of-2 bitwidths. Hence, there is often a gap between the size of a ring element and the corresponding C++ data-type e.g., `uint64_t`. In some cases, this gap can be quite large, e.g., secure inference for BERT-large communicates ring elements with bitwidths 50 in linear layers, 44 in GeLU, and 39 in Softmax. Therefore, we *pack* elements before transmitting them over the network to achieve significant communication savings over a naïve implementation that transmits standard data-types[7]. For example, we reduce communication by 35% for BERT-large.

We provide kernels for packing and unpacking elements of arbitrary bitwidths on the GPU as a part of Sigma. For packing, we make each GPU thread responsible for writing a segment of 8 bytes of data. It uses the size of the ring elements it needs to communicate to fetch the elements that belong to its segment. It also performs any shifts necessary to accommodate 'partial' elements in its segment (e.g. to pack only the first 8 bits of an element). This allows us to ensure that the packing is tight.

Since packing and unpacking require additional computation, we are implicitly trading lower communication for more computation. We find that GPUs can effectively handle this additional computation due to their high degree of parallelism. However, the cost of packing and unpacking values on CPUs overshadows the benefit of lower communication. Therefore, we do not use this optimization in our CPU implementation.

Table 1: Number of scalar activations (GeLU in BERT and GPT, SiLU in Llama2), 128-length Softmax, scalar reciprocal square roots, blocks, attention heads h and embedding length d_{model} for transformers.

Model	# Activation	# Softmax	# Rsqrt	# blocks	h	d_{model}
BERT-tiny	131072	512	512	2	2	128
BERT-base	4718592	18432	3072	12	12	768
BERT-large	12582912	49152	6144	24	16	1024
GPT-2	4718592	18432	3072	12	12	768
GPT-Neo	25165824	49152	6144	24	16	2048
Llama2-7B	45088768	131072	8192	32	32	4096
Llama2-13B	70778880	204800	10240	40	40	5120

6.0.2 CPU. The CPU code is written with 7500 lines of C++ and uses OMP for multithreading, Eigen [33] for matrix multiplications, and `cryptoTools` [66] for PRG implementations that use native x86 AES instructions.

6.0.3 SyTorch Frontend. We also develop SyTorch, a C++-based frontend, for specifying the architecture of machine learning models to be used for secure inference. It allows users to express models in a PyTorch-like high-level description and run them with various *backends*, e.g., fixed-point cleartext or Sigma's protocols for CPUs/GPUs. We provide a sample SyTorch code snippet in Figure 13 (Appendix H). Given a SyTorch model and an input, the outputs from all backends are bitwise equivalent.

The SyTorch program is compiled to a control flow graph (CFG), which is automatically transformed, e.g., relevant truncations are inserted and effective bitwidths are set (Section 5.4). The final optimized graph is then interpreted. For each operation occurring in the graph, the corresponding protocol is executed.

Table 2: Sigma has lower computation, communication, and key size than Grotto [69].

	AES or half-PRG		Comm. (Bytes)		Key Size (KB)	
	Grotto	Sigma	Grotto	Sigma	Grotto	Sigma
GeLU (CPU)	753	78	320	58	1.97	1.43
SiLU (CPU)	1087	90	320	58	1.97	1.46
Inverse	1092	254	320	36	1.97	0.17
Rsqrt	4215	1840	320	106	1.97	1.93

7 EVALUATION

We provide empirical results to justify the following claims. Sigma's protocols for complex non-linearities are up to 12× more efficient than (FSS-based) Grotto [69] (Table 2) and up to 38× more efficient than CrypTen [45] (Table 3). For end-to-end evaluation of transformers, CrypTen [45] is our primary baseline.

CrypTen is the state-of-the-art that supports the operations present in transformers, works in the 2PC with preprocessing model, and provides GPU-accelerated implementations. Note that SIGMA provides standard 2PC security whereas Crypten is insecure due to local truncations. Even so, Sigma is 11.5 – 19.4× faster than CrypTen and requires 8.4 – 11.6× lower communication for small models (Table 4). On larger models (such as Llama2-7B), CrypTen runs out of GPU memory. In contrast, Sigma scales efficiently with the number of model parameters, running inference on Llama2-7B and Llama2-13B in 27 and 44 seconds respectively.

We observe that Sigma running on CPUs is already faster than CrypTen running on GPUs. Furthermore, Sigma on GPUs is up to an order of magnitude faster than Sigma running on CPU (Figure 9). Finally, we show that Sigma's improvements over CrypTen extend to the WAN setting as well (Appendix I), with Sigma beating CrypTen by 9 – 13×.

Another potential baseline to compare against is MPCFormer [49]; we explain why such a comparison is unfair to Sigma in Appendix J.

7.0.1 Models and datasets. We evaluate BERT-tiny, BERT-base, and BERT-large models [75] on the SST2, QNLI, and MRPC classification tasks from GLUE benchmark [80]. These models have 4.4 million, 110 million, and 330 million parameters respectively. The prior work of Iron [36] also considers these models and datasets. We evaluate GPT-2 with 124 million parameters from HuggingFace (downloaded 24 million times within the last month) on the challenging Lambada dataset [58], which has next-word-prediction tasks. For billion parameter models, we evaluate GPT-Neo-1.3B, LLaMa2-7B and LLaMa2-13B from HuggingFace [3–5], also on Lambada. These models use GeLU, SiLU and Softmax in abundance (Table 1). We also report the number of reciprocal square roots arising because of layer normalizations. Prior works have observed that these non-linearities are the performance bottlenecks in secure inference of transformers [36, 49]. Following Iron [36], we evaluate all models on inputs of sequence length 128. We evaluate other sequence lengths in Appendix K. We set the precision $f = 12$, and the bitwidths to be large enough so that Sigma's accuracy matches that of 32-bit floating-point PyTorch (Appendix L). In particular, for BERT-tiny a bitwidth of 37 suffices, whereas the other models require larger bitwidths (between 48 and 51).

7.0.2 Hardware. We evaluate on two machines connected via LAN with 9.4 Gbps bandwidth and 0.05 ms ping time. Each machine has 1 TB RAM, an A6000 GPU with 46GB GPU memory, and an AMD Epyc 7742 processor. Sigma running on CPUs uses 4 threads.

7.1 Non-linearities

We show our performance improvements for GeLU, SiLU, Softmax, and layer normalization over the baselines.

7.1.1 Comparison with Grotto. Grotto [69] is a recent work that provides FSS-based protocols for GeLU, SiLU, inverse (that arises in Softmax), and reciprocal square root (that arises in layer normalization). Table 2 shows that, for each of these functions, Sigma beats Grotto in all aspects: computation, communication, and key size. Since the source code of Grotto is unavailable, we cannot evaluate it on our setup. However, the communication and the key size are independent of the setup. The compute cost of FSS-based protocols like Grotto and Sigma is heavily dominated by PRG calls, and we use these as a proxy for the computation overheads.

7.1.2 Comparison with Orca. Orca [41] is the state-of-the-art in GPU-accelerated FSS and it proposes the recipe of using 2PC floating-point protocols [62] for complex non-linearities like Softmax. The communication overheads of this approach are severe – requiring 7 GB (for BERT-tiny) to 1.1 TB (for GPT-Neo) of communication for evaluating GeLU and Softmax. In contrast, Sigma's communication is between 20 MB and 4 GB (Table 4) for these models.

7.1.3 Comparison with CrypTen. We compare Sigma (both CPU and GPU implementations) and CrypTen by measuring their latency and communication in evaluating activations (GeLU/SiLU), Softmax and LayerNorm (Table 3). For GeLU and Softmax, Sigma's communication is an order of magnitude lower than CrypTen. Due to this, Sigma's protocols running on CPUs outperform CrypTen on GPUs on all transformers. For LayerNorm, CrypTen's communication is low because of its use of local truncation. However, our protocol for reciprocal square root is more efficient and our runtimes for LayerNorm on CPUs are 2.6 – 20× better. Furthermore, with GPU acceleration, Sigma outperforms CrypTen by at least 10× for all three non-linearities on all transformers. Finally, the lower communication of Sigma running on GPUs (vs. CPUs) is due to communication packing (Section 6).

7.2 Transformers

We evaluate Sigma on end-to-end transformer inference to show that it beats CrypTen in latency and communication. We also show that GPU acceleration is helpful for Sigma, and that Sigma scales well to larger models. Preprocessing costs are not included for both CrypTen and Sigma. CrypTen does not report preprocessing cost, and we describe Sigma's preprocessing cost in Appendix M. From Tables 4 and 8, we see that Sigma's *total* time, which includes preprocessing, is 1.3 – 5× lower than just CrypTen's online time.

[7]While Orca packs 1 or 2-bit values, we support packing for *all* non-powers-of-2 bitwidths in Sigma, providing benefit in all our protocols. While reporting improvements, we use the baseline that packs 1 or 2-bit values but uses standard data-types for rest.

Table 3: Sigma outperforms CrypTen (GPU) on Activation (GeLU in BERT and GPT, SiLU in LLAMA), Softmax and Norm (LayerNorm in BERT and GPT, RMSNorm in Llama2). CT denotes CrypTen, and S-CPU and S-GPU stand for Sigma running on CPU and GPU respectively. "-" denotes GPU memory overflow.

Model	Activation (GeLU / SiLU)						Softmax						Norm (LayerNorm / RMSNorm)					
	Time (s)			Communication (GB)			Time (s)			Communication (GB)			Time (s)			Communication (GB)		
	CT	S-CPU	S-GPU	CT	S-CPU	S-GPU	CT	S-CPU	S-GPU	CT	S-CPU	S-GPU	CT	S-CPU	S-GPU	CT	S-CPU	S-GPU
BERT-tiny	0.27	0.06	0.007	0.10	0.01	0.003	0.71	0.09	0.02	0.09	0.01	0.005	0.60	0.03	0.03	0.003	0.004	0.002
BERT-base	4.59	3.76	0.25	3.45	0.25	0.16	7.53	4.42	0.44	3.27	0.37	0.26	4.31	0.67	0.25	0.11	0.15	0.11
BERT-large	11.50	9.84	0.66	9.19	0.66	0.42	17.35	11.94	1.13	8.72	1.00	0.69	8.75	1.78	0.55	0.29	0.40	0.30
GPT-2	4.47	3.76	0.25	3.45	0.25	0.16	6.89	2.76	0.27	3.27	0.19	0.13	3.94	0.69	0.25	0.11	0.15	0.11
GPT-Neo	20.35	20.35	1.33	18.38	1.69	0.86	16.33	7.55	0.66	8.72	0.50	0.36	8.91	3.39	0.80	0.57	0.80	0.60
Llama2-7B	-	51.19	5.09	-	2.50	1.47	-	22.97	1.45	-	1.37	0.90	-	4.81	1.59	-	1.60	1.52
Llama2-13B	-	78.03	8.04	-	3.84	2.31	-	38.85	2.29	-	2.10	1.40	-	7.19	2.72	-	2.44	2.37

Table 4: Sigma vs CrypTen on end-to-end inference. "-" denotes GPU memory overflow.

Model	Time (s)			Communication (GB)	
	CrypTen	Sigma	Speedup	CrypTen	Sigma
BERT-tiny	1.71	0.09	19.4×	0.20	0.02
BERT-base	21.55	1.84	11.7×	8.34	0.99
BERT-large	54.53	4.73	11.5×	23.36	2.63
GPT-2	20.45	1.61	12.7×	8.34	0.82
GPT-Neo	108.30	7.43	14.6×	46.89	4.02
Llama2-7B	-	27.01	-	-	12.35
Llama2-13B	-	44.13	-	-	19.33

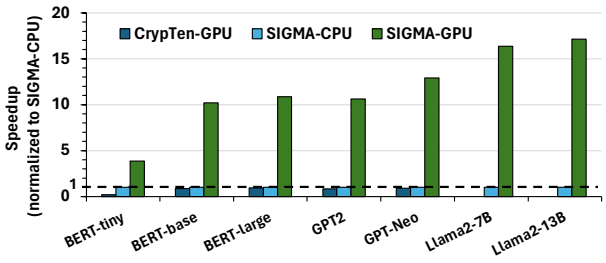

Figure 9: Sigma GPU and CrypTen speedups over Sigma CPU.

7.2.1 Comparison with CrypTen. Table 4 shows the performance of various transformer models with CrypTen and Sigma, both running on GPUs. There are two factors here: 1) Sigma uses secure but more expensive (in compute and communication) truncations than CrypTen's local truncations, and 2) Sigma's protocols for non-linearities have massive improvements over CrypTen (Section 7.1.3). Overall, for end-to-end inference, Sigma outperforms CrypTen by 11.5 – 19.4× in latency and 8.4 – 11.6× in communication.

7.2.2 GPU acceleration. Figure 9 shows the speedups of CrypTen and Sigma running on GPUs over Sigma running on CPUs. For end-to-end transformer inference, Sigma running on CPUs is always faster than CrypTen running on GPUs. Sigma's protocols for GPUs are an order of magnitude faster compared to their CPU counterparts for all models except BERT-tiny, which is too small to leverage GPUs effectively.

7.2.3 Scaling to larger models. Sigma scales efficiently to large models. Figure 10 shows the runtime of Sigma as the number of model parameters increases. We use GPT-Neo, Llama2-7B and

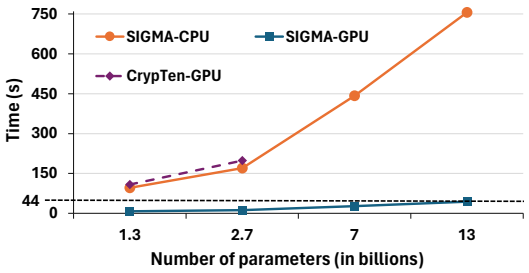

Figure 10: Sigma scaling to larger models. CrypTen runs out of memory for 7B and 13B models.

Llama2-13B as our 1.3, 7 and 13 billion parameter models respectively. We create a GPT-like 2.7 billion parameter model by increasing the number of blocks and attention heads in GPT-Neo to 20 and 32. We see that Sigma running on GPUs performs inference on the 13 billion parameter LLaMa2 model in 44 seconds. In contrast, CrypTen overflows GPU memory on the 7 billion and 13 billion parameter models and crashes with an out-of-memory exception.

8 RELATED WORK

Secure inference (with MPC or with other techniques like TEEs [74] or FHE [30]) has a vast literature and we don't attempt to survey it. Here, we focus on works related to transformers, GPU acceleration of MPC, and FSS.

After the success of large models like GPT3/GPT3.5 with 175 billion parameters, there are ongoing efforts to reduce the cost and latency of inference by using smaller models [8, 71, 72]. For example, phi-1 outperforms GPT-3.5 models while using only 1.3 billion parameters [34]. Another approach to reduce the latency of secure inference involves replacing complex non-linearities that are expensive in MPC with simple non-linearities. The simple approximations significantly impact accuracy but, at least for BERT class models, this accuracy loss can be recovered by further retraining of the simplified models [53]. THE-X [22], MPCformer [49] and Privformer [9] use simple non-linearities. In contrast, Iron [36], CrypTen [45], and Sigma use precise approximations of complex non-linearities and there is no accuracy loss. Recent pipelining optimizations have improved the performance of CrypTen by up to 13% [81] and such optimizations can benefit Sigma as well.

There are several works that focus on accelerating secure inference with GPUs, but to support CNNs and not transformers. CryptGPU accelerates 3-party secure inference with GPUs [70]. Piranha is a general framework that supports various number of parties [82]. Delphi performs a network architecture search to navigate performance-accuracy tradeoffs. GForce uses custom training approaches to improve inference efficiency [57]. Beyond inference, Visor [60] focuses on video analytics and general protocols like Yao's garbled circuits have also been accelerated with GPUs [39].

Several recent works consider 2PC in the preprocessing model based on FSS techniques. [16] initiated this study and showed how to construct 2PC protocols for any computation comprising of gates for which FSS constructions exist for the corresponding offset gate. [13] provides various FSS protocols for functions occurring in fixed-point arithmetic, while [35, 68, 77, 83] provides specialized FSS protocols for ML operations. [68] and [41] accelerate FSS protocols on GPUs while [7] and [69] consider FSS protocols for various elementary functions such as sigmoid, GeLU, and so on.

9 CONCLUSION

We build Sigma, the first system for FSS-based secure inference of transformers. To this end, we build novel protocols for GeLU, SiLU, Softmax, and layer normalization. The same techniques generalize to construct efficient protocols for other activations such as sigmoid, CELU, etc. Sigma satisfies standard 2PC security, matches PyTorch accuracy, and is an order of magnitude faster than the baselines. Similar to all prior works on secure inference of transformers, Sigma focuses on semi-honest security and we leave security against malicious adversaries [21, 23, 28, 29, 43, 48] for future work.

ACKNOWLEDGEMENTS

We thank Tanmay Rajore for his help in implementing the secure protocol for converting fixed point numbers to floating point.

REFERENCES

[1] 2023. CUTLASS. https://github.com/NVIDIA/cutlass
[2] 2023. GPT-2. https://huggingface.co/gpt2.
[3] 2023. GPT Neo. https://huggingface.co/docs/transformers/model_doc/gpt_neo.
[4] 2023. Meta Llama2 13B. https://huggingface.co/meta-llama/Llama-2-13b.
[5] 2023. Meta Llama2 7B. https://huggingface.co/meta-llama/Llama-2-7b.
[6] 2023. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.
[7] Amit Agarwal, Stanislav Peceny, Mariana Raykova, Phillipp Schoppmann, and Karn Seth. 2022. Communication Efficient Secure Logistic Regression. *IACR Cryptol. ePrint Arch.* (2022), 866. https://eprint.iacr.org/2022/866
[8] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Guiding Language Models of Code with Global Context using Monitors. arXiv:2306.10763 [cs.CL]
[9] Y. Akimoto, K. Fukuchi, Y. Akimoto, and J. Sakuma. 2023. Privformer: Privacy-Preserving Transformer with MPC. In *EuroS&P*.
[10] Donald Beaver. '91. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*.
[11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. 2011. Semi-homomorphic Encryption and Multiparty Computation. In *EUROCRYPT*.
[12] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. 2020. MP2ML: a mixed-protocol machine learning framework for private inference. In *ARES*.
[13] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. 2020. Function Secret Sharing for Mixed-Mode and Fixed-Point Secure Computation. In *EUROCRYPT*.
[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2015. Function secret sharing. In *EUROCRYPT*.
[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function Secret Sharing: Improvements and Extensions. In *CCS*.
[16] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2019. Secure Computation with Preprocessing via Function Secret Sharing. In *TCC*.
[17] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]
[18] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL]
[19] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology* (2000).
[20] Nicholas Carlini, Matthew Jagielski, and Ilya Mironov. 2020. Cryptanalytic Extraction of Neural Network Models. In *Advances in Cryptology – CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III* (Santa Barbara, CA, USA). Springer-Verlag, Berlin, Heidelberg, 189–218. https://doi.org/10.1007/978-3-030-56877-1_7
[21] Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. 2022. SIMC: ML Inference Secure Against Malicious Clients at Semi-Honest Cost. In *USENIX Security Symposium*.
[22] Tianyu Chen, Hangbo Bao, Shaohan Huang, Li Dong, Binxing Jiao, Daxin Jiang, Haoyi Zhou, Jianxin Li, and Furu Wei. 2022. THE-X: Privacy-Preserving Transformer Inference with Homomorphic Encryption. In *ACL*.
[23] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. 2020. Secure Evaluation of Quantized Neural Networks. *PoPETs* (2020).
[24] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. 2017. The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited. In *CRYPTO*.
[25] Ivan Damgård and Sarah Zakarias. 2013. Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing. In *TCC*.
[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]
[27] Jack Doerner and Abhi Shelat. 2017. Scaling ORAM for Secure Computation. In *CCS*.
[28] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO*.
[29] Tore Kasper Frederiksen, Thomas P. Jakobsen, and Jesper Buus Nielsen. 2014. Faster Maliciously Secure Two-Party Computation Using the GPU. In *SCN*.
[30] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *ICML*.
[31] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*.
[32] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *PLDI*.
[33] Gaël Guennebaud and Benoît Jacob. 2010. Eigen v3. http://eigen.tuxfamily.org.
[34] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks Are All You Need. arXiv:2306.11644
[35] Kanav Gupta, Deepak Kumaraswamy, Nishanth Chandran, and Divya Gupta. 2022. LLAMA: A Low Latency Math Library for Secure Inference. In *PETS*.
[36] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. 2022. Iron: Private Inference on Transformers. In *NeurIPS*.
[37] Dan Hendrycks and Kevin Gimpel. 2016. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *CoRR* abs/1606.08415 (2016). arXiv:1606.08415 http://arxiv.org/abs/1606.08415
[38] Zhicong Huang, Wen jie Lu, Cheng Hong, and Jiansheng Ding. 2022. Cheetah: Lean and Fast Secure Two-Party Deep Neural Network Inference. In *USENIX Security Symposium*.
[39] Nathaniel Husted, Steven Myers, Abhi Shelat, and Paul Grubbs. 2013. GPU and CPU Parallelization of Honest-but-Curious Secure Two-Party Computation. In *ACSAC*.
[40] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. 2013. On the Power of Correlated Randomness in Secure Computation. In *TCC*.
[41] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2023. Orca: FSS-based Secure Training with GPUs. Cryptology ePrint Archive, Paper 2023/206.

[42] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium*.
[43] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS*.
[44] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-Normalizing Neural Networks. arXiv:1706.02515 [cs.LG]
[45] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubhabrata Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *NeurIPS*.
[46] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow: Secure TensorFlow Inference. In *IEEE S&P*.
[47] Maximilian Lam, Jeff Johnson, Wenjie Xiong, Kiwan Maeng, Udit Gupta, Minsoo Rhu, Hsien-Hsin S. Lee, Vijay Janapa Reddi, Gu-Yeon Wei, David Brooks, and Edward Suh. 2023. GPU-based Private Information Retrieval for On-Device Machine Learning Inference. *CoRR* abs/2301.10904 (2023).
[48] Ryan Lehmkuhl, Pratyush Mishra, Akshayaram Srinivasan, and Raluca Ada Popa. 2021. Muse: Secure Inference Resilient to Malicious Clients. In *USENIX Security Symposium*.
[49] Dacheng Li, Hongyi Wang, Rulin Shao, Han Guo, Eric Xing, and Hao Zhang. 2023. MPCFORMER: FAST, PERFORMANT AND PRIVATE TRANSFORMER INFERENCE WITH MPC. In *ICLR*.
[50] Yun Li, Yufei Duan, Zhicong Huang, Cheng Hong, Chao Zhang, and Yifan Song. 2023. Efficient 3PC for Binary Circuits with Application to Maliciously-Secure DNN Inference. In *USENIX Security Symposium*.
[51] Yehuda Lindell. 2017. How to Simulate It – A Tutorial on the Simulation Proof Technique. *Tutorials on the Foundations of Cryptography* (2017).
[52] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *CCS*.
[53] Neo Wei Ming, Zhehui Wang, Cheng Liu, Rick Siow Mong Goh, and Tao Luo. 2023. MA-BERT: Towards Matrix Arithmetic-only BERT Inference by Eliminating Complex Non-Linear Functions. In *ICLR*.
[54] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *USENIX Security Symposium*.
[55] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*.
[56] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.
[57] Lucien K. L. Ng and Sherman S. M. Chow. 2021. GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference. In *USENIX Security Symposium*.
[58] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernandez. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *ACL*.
[59] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security Symposium*.
[60] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. 2020. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *USENIX Security Symposium*.
[61] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2017. Searching for Activation Functions. arXiv:1710.05941 [cs.NE]
[62] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. 2022. SecFloat: Accurate Floating-Point meets Secure 2-Party Computation. In *IEEE S&P*.
[63] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SIRNN: A math library for secure inference of RNNs. In *IEEE S&P*.
[64] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. 2020. CrypTFlow2: Practical 2-Party Secure Inference. In *CCS*.
[65] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ASIACCS*.
[66] Peter Rindal. 2023. cryptoTools. https://github.com/ladnir/cryptoTools.
[67] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. 2018. DeepSecure: Scalable Provably-Secure Deep Learning. In *DAC*.
[68] Théo Ryffel, David Pointcheval, and Francis Bach. 2022. ARIANN: Low-Interaction Privacy-Preserving Deep Learning via Function Secret Sharing. In *PETS*.
[69] Kyle Storrier, Adithya Vadapalli, Allan Lyons, and Ryan Henry. 2023. Grotto: Screaming fast $(2+1)$-PC for $\mathbb{Z}_{2^n}$ via (2, 2)-DPFs. Cryptology ePrint Archive, Paper 2023/108.
[70] Sijun Tan, Brian Knott, Yuan Tian, and David J. Wu. 2021. CryptGPU: Fast Privacy-Preserving Machine Learning on the GPU. In *IEEE S&P*.
[71] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
[72] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]
[73] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]
[74] Florian Tramèr and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *ICLR*.
[75] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation. *CoRR* abs/1908.08962 (2019).
[76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*.
[77] Sameer Wagh. 2022. Pika: Secure Computation using Function Secret Sharing over Rings. *PoPETs* (2022).
[78] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *PoPETs* (2019).
[79] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PoPETs* (2021).
[80] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *ICLR*.
[81] Yongqin Wang, Rachit Rajat, and Murali Annavaram. 2022. MPC-Pipe: an Efficient Pipeline Scheme for Secure Multi-party Machine Learning Inference. *CoRR* abs/2209.13643 (2022).
[82] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. 2022. Piranha: A GPU Platform for Secure Computation. In *USENIX Security Symposium*.
[83] Peng Yang, Zoe L. Jiang, Shiqi Gao, Jiehang Zhuang, Hongxiao Wang, Junbin Fang, Siuming Yiu, and Yulin Wu. 2023. FssNN: Communication-Efficient Secure Neural Network Training via Function Secret Sharing. Cryptology ePrint Archive, Paper 2023/073.
[84] Andrew Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *FOCS*.

A REAL WORLD CONSIDERATIONS

A.0.1 Generality. This work focuses on transformers as it is the most important AI workload today. However, our work also generalizes to other networks that may use activations such as Softmax, GeLU, etc. These functions are known to be expensive under various secure computation settings. For example, [41] reports that in the secure training of some CNNs, Softmax accounts for 92% of the training runtime. Sigma's fast, accuracy-preserving approximations of these functions can ameliorate such performance bottlenecks.

A.0.2 Practicality. Secure inference is much more expensive than plaintext inference. For example, cleartext inference with LLAMA-7B using PyTorch takes 0.37 seconds, while Sigma takes 27 seconds, a gap of 73×. This gap is prevalent across networks. For CNNs, PyTorch takes 2.4ms for cleartext VGG16 inference on ImageNet, while Orca [41], the current state-of-the-art in the secure inference of CNNs, takes 0.5s (a gap of more than 200×). We believe that

there are several use-cases where such an overhead is acceptable to provide strong privacy guarantees.

A.0.3 Preventing Model Extraction. Model extraction attacks try to infer model weights given access to model inputs and the corresponding outputs. In secure inference, the client has access to inputs *and* outputs in the clear and can potentially try to mount a model extraction attack. Circumventing such attacks is orthogonal to the entire line of work on secure inference that only deals with computing functions without trusting anyone with cleartext inputs. How the outputs of the function are used (even if adversarially) is an independent line of investigation. It is to be noted that such model extraction attacks are known only for very simple networks [20].

B FSS CORRECTNESS AND SECURITY

Definition 2 (FSS: Correctness and Security [14, 15]). *Let $\mathcal{G} = \{g\}$ be a function family, $P_{\mathcal{G}} = \{\hat{g}\}$ be the set of descriptions of functions in $\mathcal{G}$, and* Leak *be a function specifying the allowable leakage about $\hat{g}$. When* Leak *is omitted, it is understood to output only $\mathbb{G}^{\mathsf{in}}$ and $\mathbb{G}^{\mathsf{out}}$. We say that* (Gen, Eval) *as in Definition 1 is an FSS scheme for $\mathcal{G}$ (with respect to leakage* Leak*) if it satisfies the following.*

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

C PROOF OF LEMMA 2

To calculate $1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\}$, consider four cases:

(1) *Case 1:* $\mathsf{MSB}_n(\hat{x}) = 1$ *and* $\mathsf{MSB}_n(\mathsf{r}^{in}) = 0$.
Since $\hat{x} \geq 2^{n-1} > \mathsf{r}^{\mathsf{in}}$, $1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} = 0$ follows trivially.

(2) *Case 2:* $\mathsf{MSB}_n(\hat{x}) = 0$ *and* $\mathsf{MSB}_n(\mathsf{r}^{in}) = 1$.
Since $\hat{x} < 2^{n-1} \leq \mathsf{r}^{\mathsf{in}}$, $1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} = 1$ follows trivially.

(3) *Case 3:* $\mathsf{MSB}_n(\hat{x}) = \mathsf{MSB}_n(\mathsf{r}^{in}) = 0$.
As $x < 2^{n-1}$ and $\mathsf{r}^{\mathsf{in}} < 2^{n-1}$, $x + \mathsf{r}^{\mathsf{in}} < 2^n \implies \hat{x} = x + \mathsf{r}^{\mathsf{in}} \mod 2^n = x + \mathsf{r}^{\mathsf{in}} \geq \mathsf{r}^{\mathsf{in}} \implies 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} = 0$.

(4) *Case 3:* $\mathsf{MSB}_n(\hat{x}) = \mathsf{MSB}_n(\mathsf{r}^{in}) = 1$.
As $x < 2^{n-1}$ and $2^{n-1} \leq \mathsf{r}^{\mathsf{in}} < 2^n$, $x + \mathsf{r}^{\mathsf{in}} \in [2^{n-1}, 2^n + 2^{n-1})$. But as $\hat{x} \geq 2^{n-1}$, $x + \mathsf{r}^{\mathsf{in}} < 2^n$. Hence, $\hat{x} = x + \mathsf{r}^{\mathsf{in}} \mod 2^n = x + \mathsf{r}^{\mathsf{in}} \geq \mathsf{r}^{\mathsf{in}} \implies 1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} = 0$.

Hence, $1\{\hat{x} < \mathsf{r}^{\mathsf{in}}\} = 1\{\mathsf{MSB}_n(\hat{x}) = 0 \text{ and } \mathsf{MSB}_n(\mathsf{r}^{\mathsf{in}}) = 1\} = \mathsf{MSB}_n(\mathsf{r}^{\mathsf{in}}) \cdot (1 - \mathsf{MSB}_n(\hat{x}))$.

D SECURITY PROOFS

Let $\mathsf{Sim}_n^<$ be the simulator for the FSS-scheme of comparison function from Theorem 2. As we use $\mathsf{Gen}_n^\bullet$ from [15] directly in this FSS-scheme, Definition 2 implies that the security of the FSS-scheme for comparison trivially follows from the security of DPF construction of [15].

D.1 DReLU

For $b \in \{0,1\}$, let $\mathsf{Sim}_b^{\mathsf{DReLU}}$ be the simulator for the protocol Π_n^{DReLU}. It is given the input $\hat{x} \in \mathbb{U}_N$ and output $u_b \in \{0,1\}$. It simulates the view of party b, by simulating the message $r_b || k_b^\bullet$ from dealer by following these steps:

(1) Set $\hat{y} = \hat{x} \mod 2^{n-1}$
(2) Invoke $\mathsf{Sim}_n^<$ to simulate the DPF key $k_{b,\mathsf{sim}}^\bullet$
(3) Set $t_{b,\mathsf{sim}} \leftarrow \mathsf{Eval}_{n-1}^<(b, k_{b,\mathsf{sim}}^\bullet, 2^{n-1} - \hat{y} - 1)$
(4) Set $r_{b,\mathsf{sim}} = b \cdot \mathsf{MSB}_n(\hat{x}) \oplus u_b \oplus t_{b,\mathsf{sim}}$.
(5) Output $r_{b,\mathsf{sim}} || k_{b,\mathsf{sim}}^\bullet$.

D.2 LRS with Gap

For $b \in \{0,1\}$, let $\mathsf{Sim}_b^{\mathsf{GapLRS}}$ be the simulator for the protocol $\Pi_{n,f}^{\mathsf{GapLRS}}$. It is given the input $\hat{x} \in \mathbb{U}_N$ and output $y_b \in \mathbb{U}_N$. It simulates the view of party b, by simulating the message $k_b^\bullet || r_b^{(w)} || m_b || r_b$ from dealer and $\hat{w}_{1-b}$ from the other party, by following these steps:

(1) Sample $r_{b,\mathsf{sim}}^{(w)}, \hat{w}_{1-b,\mathsf{sim}} \xleftarrow{\$} \{0,1\}$.
(2) Invoke $\mathsf{Sim}_f^<$ to simulate DPF keys $k_{b,\mathsf{sim}}^\bullet$
(3) Set $\hat{w}_{b,\mathsf{sim}} = \mathsf{Eval}_f^<(b, k_{b,\mathsf{sim}}^\bullet, \hat{x} \mod 2^f) \oplus r_{b,\mathsf{sim}}^{(w)}$
(4) Set $\hat{w}_{\mathsf{sim}} = \hat{w}_{b,\mathsf{sim}} \oplus \hat{w}_{1-b,\mathsf{sim}}$, $\hat{z}_{\mathsf{sim}} = \mathsf{extend}_{1,n}(\hat{w}_{\mathsf{sim}})$
(5) Set $u_{b,\mathsf{sim}} = b\hat{z}_{\mathsf{sim}} + r_{b,\mathsf{sim}}^{(w)} - 2\hat{z}_{\mathsf{sim}} r_{b,\mathsf{sim}}^{(w)}$
(6) Sample $m_{b,\mathsf{sim}} \xleftarrow{\$} \mathbb{U}_N$.
(7) Set $t_{b,\mathsf{sim}} = m_{b,\mathsf{sim}} \cdot \mathsf{extend}_{1,n}(1 - \mathsf{MSB}_n(\hat{x}))$
(8) Set $r_{b,\mathsf{sim}} = y_b - b \cdot \mathsf{LRS}_{n,f}(\hat{x}) - t_{b,\mathsf{sim}} + u_{b,\mathsf{sim}}$
(9) Output $k_{b,\mathsf{sim}}^\bullet || r_{b,\mathsf{sim}}^{(w)} || m_{b,\mathsf{sim}} || r_{b,\mathsf{sim}}$ and $\hat{w}_{1-b,\mathsf{sim}}$.

E SILU

For a real number x, $\mathsf{SiLU}(x) = x \,.\, \sigma(x) = \frac{x}{1+e^{-x}}$, where $\sigma(x)$ is the Sigmoid function. Like GeLU, SiLU is the same as ReLU almost everywhere except in a small interval around 0. We define $\delta(x) = \mathsf{ReLU}(x) - \mathsf{SiLU}(x)$ and compute $\mathsf{SiLU}(x)$ as $\mathsf{ReLU}(x) - \delta(x)$. $\delta(x)$ is even (as it was for GeLU), and, for precision $f = 12$, is zero outside the interval $(-16, 16)$.

Our approximation of SiLU follows from our approximation of GeLU, with a few small differences. The clipping interval (A, B) is now $[-2^{f+4} + 1, 2^{f+4} - 1]$. As a result, the LUT for $\delta(x)$ is larger, and has $2^{10} = 1024$ entries instead of $2^8 = 256$ entries as it did for GeLU. Correspondingly, our CPU and GPU protocols for SiLU are also similar to those for GeLU, while accounting for the above differences.

F OTHER ACTIVATIONS

Our approach to constructing efficient yet sufficiently precise approximations of GeLU and SiLU can be extended to other activations as well. Consider the Continuously Differentiable Exponential Linear Unit, or CELU, defined as $\mathsf{CELU}(x) = max(0, x) + min(0, e^x - 1)$ [69]. To approximate CELU, we define a piecewise-linear function $f(x)$ as

$$f(x) = \begin{cases} -1 & x \leqslant 0 \\ x & x > 0 \end{cases}$$

$f(x)$ is similar to ReLU and can be realized with our protocols for DReLU and a slightly modified version of $\mathsf{selectlin}_{n,\gamma}$, where γ is now a pair of 2-vectors $\{(\alpha_0,\beta_0),(\alpha_1,\beta_1)\}$, that takes as input a selection bit s and a payload x and outputs $\alpha_s x + \beta_s$. Setting $(\alpha_0,\beta_0) = (0,-1)$, $(\alpha_1,\beta_1) = (1,0)$ and using $\mathsf{DReLU}(x)$ as the selection bit with x as the payload allows us to realize f. We define $\delta(x) = \mathsf{CELU}(x) - f(x)$, which is 0 when $x > 0$ and e^x when $x \leqslant 0$. $\mathsf{CELU}(x)$ is naturally computed as $f(x)+\delta(x)$. For precision $f = 12$, $\delta(x)$ disappears outside $(-16, 0)$, and can thus be realized with an LUT (of size 1024) after appropriate clipping of the input. The above protocol can be adapted to compute an approximation of the more general Scaled Exponential Linear Unit, or SELU[44] as well.

We can also design an approximation of the pervasive Sigmoid activation, defined as $\sigma(x) = \frac{1}{1+e^{-x}}$. We start by setting $\delta(x) = \frac{1}{1+e^{|x|}}$ and realizing that

$$\sigma(x) = \begin{cases} \delta(x) & x \leqslant 0 \\ 1-\delta(x) & x > 0 \end{cases}$$

$\delta(x)$ disappears outside $(-16, 16)$ for precision $f = 12$ and can be realized with an LUT (of size 1024, since it is even). We can then use the modified selectlin defined above with $(\alpha_0,\beta_0) = (1,0)$, $(\alpha_1,\beta_1) = (-1,1)$, with $\mathsf{DReLU}(x)$ as the selection bit and $\delta(x)$ as the payload, to get $\sigma(x)$.

G LAYER NORMALIZATION

The functionality of layer normalization, as defined in Section 5.3, calls reciprocal square root with variance of the input vector as an input. Our protocol for reciprocal square root (Appendix G.2) makes use of the protocol for interval lookup (Appendix G.1). Finally, we provide the overall optimized protocol for layer normalization in Appendix G.3.

G.1 Interval Lookup

Let $\boldsymbol{p},\boldsymbol{q} \in \mathbb{U}_N^k$ be arrays defining k disjoint intervals $[p[i], q[i]]\forall i \in [k]$, constrained with $p[i+1] = q[i]\forall i \in [k-1]$, $p[0] = 0$ and $q[k-1] = 2^n - 1$. Let $\boldsymbol{v} \in \mathbb{U}_L^k$ be a payload array. We define the functionality $\mathsf{IntervalLookup}_{n,\mathbb{U}_L,\boldsymbol{p},\boldsymbol{q},\boldsymbol{v}} : \mathbb{U}_N \to \mathbb{U}_L$ which returns $v[i]$ when $x \in [p[i], q[i]]$ for some $i \in [k]$. Since this functionality is equivalent to a 0-degree spline, we use the protocol for splines from Grotto [69] to implement this. Even though the protocol invokes DPF evaluation k times, they significantly reduce the number of half PRG calls compared to nk using the memoization technique, which caches the intermediate seeds in DPF tree to be reused in subsequent evaluations. We omit details and directly summarize the costs of the protocol:

Theorem 7. *Let $\ell = \lceil \log_2(|\mathbb{G}|) \rceil$ and $\boldsymbol{p},\boldsymbol{q} \in \mathbb{U}_N^k, \boldsymbol{v} \in \mathbb{G}^k$ be arrays of size k. There exists a protocol $\Pi^{\mathsf{IntervalLookup}}_{n,\mathbb{G},\boldsymbol{p},\boldsymbol{q},\boldsymbol{v}}$ which securely realizes $\mathsf{IntervalLookup}_{n,\mathbb{G},\boldsymbol{p},\boldsymbol{q},\boldsymbol{v}}$ such that $\mathsf{keysize}(\Pi^{\mathsf{IntervalLookup}}_{n,\mathbb{G},\boldsymbol{p},\boldsymbol{q},\boldsymbol{v}}) = \mathsf{keysize}(\mathsf{DPF}_{n,1}) + 3\ell$. In the online phase, the protocol requires k memoized evaluations of $\mathsf{DPF}_{n,1}$ and communication of 4ℓ bits in 1 round.*

G.2 Reciprocal Square Root

For bitwidth n, input precision f^{in} and output precision f^{out}, we define the function $\mathsf{RecSqrt}_{n,f^{\mathsf{in}},f^{\mathsf{out}}}$ to be the approximation of the reciprocal square root of a fixed-point number $x \in \mathbb{U}_N$ with scale f^{in}. It returns a fixed-point number $y \in \mathbb{U}_N$ with scale f^{out}, i.e., $\mathsf{uint}_n(y) \approx \sqrt{2^{f^{\mathsf{in}}}/x} \cdot 2^{f^{\mathsf{out}}}$.

As discussed in Section 5.3.1, since the inputs of reciprocal square root occurring in layer normalization are unconstrained, to get a small LUT, we first convert the input to a custom floating point representation. This allows us to represent a large dynamic range using only a small number of bits. A similar protocol for converting fixed-point numbers to IEEE 32-bit floating-point numbers was provided by Orca [41].

A floating-point representation has a sign bit, exponent bits, and mantissa bits. Taking inspiration from the `bfloat16` datatype which is being extensively used in ML, we also use a 7-bit mantissa. As we are only interested in non-zero positive n-bit integers with $n \leq 64$, a 6-bit exponent suffices and we don't need a sign bit. This 13-bit index is used to look-up the fixed-point output.

Let $x \in \mathbb{U}_N$ be the input to RecSqrt. We convert the integer representation of x to float-like representation and input precision f^{in} would be handled in the LUT later. Let $m \in \mathbb{U}_{128}, e \in \mathbb{U}_{64}$ represent the mantissa and exponent of the floating point representation of x. So, it must hold that:

$$\mathsf{uint}_n(x) \approx 2^{\mathsf{uint}_6(e)} \cdot (1 + \frac{\mathsf{uint}_7(m)}{128}) \tag{4}$$

From here on, we suppress $\mathsf{uint}(\cdot)$ whenever it is clear from context. Let $k \in \mathbb{U}_{64}$ be a number such that $2^{k-1} \leq x < 2^k$. As $1 \leq (1 + m/128) < 2$, it holds that $2^e \leq 2^e \cdot (1 + m/128) < 2^{e+1}$ and hence, we can set $e = k-1$. To calculate m, we plug $e = k-1$ in Equation 4:

$$x \approx 2^{k-1} \cdot (1 + \frac{m}{128})$$
$$\implies m \approx \frac{x \cdot 128}{2^{k-1}} - 128 = \frac{x \cdot 2^{n-k}}{2^{n-8}} - 128$$

Let $u = 2^{n-k} \in \mathbb{U}_N$. As $x < 2^k$, $x \cdot 2^{n-k} < 2^n$ and can be encoded in n bits. So, we can approximate m as:

$$\begin{aligned} m &\approx \mathsf{TR}_{n,n-8}(x \cdot u) - 128 \mod 128 \\ &= \mathsf{TR}_{n,n-8}(x \cdot u) \mod 128 \end{aligned} \tag{5}$$

To securely calculate $e = k-1$ and $u = 2^{n-k}$, we can use the protocol for interval lookup (Appendix G.1). Let $\mathbb{G} = \mathbb{U}_{2^{13}} \times \mathbb{U}_{2^n}$. Let $\boldsymbol{p},\boldsymbol{q} \in \mathbb{U}_N^n, \boldsymbol{v} \in \mathbb{G}^n$ be arrays s.t. $p[0] = 0, q[0] = 1, v[0] = (0, 2^{n-1})$, and $\forall i \in [1, n-1]$:

$$p[i] = q[i-1] + 1,\ q[i] = 2^{i+1} - 1,\ v[i] = (i, 2^{n-i-1})$$

Then, it trivially holds that:

$$(\mathsf{extend}_{6,13}(e), u) = \mathsf{IntervalLookup}_{n,\mathbb{G},\boldsymbol{p},\boldsymbol{q},\boldsymbol{v}}(x)$$

Finally, we can calculate m using Equation 5 and concatenate e to get the required floating point representation as:

$$p = m||e = \mathsf{extend}_{7,13}(m) \cdot 2^6 + \mathsf{extend}_{6,13}(e)$$

Note that local extension suffices in case of m, as the result is being multiplied by 2^6, due to which wrap error vanishes. Now, we

Reciprocal Square Root $\Pi^{\mathsf{RecSqrt}}_{n,f^{\mathsf{in}},f^{\mathsf{out}}}(\hat{x})$

1: $(\hat{e}_b, \hat{u}_b) \leftarrow \Pi^{\mathsf{IntervalLookup}}_{n,\mathbb{G},\boldsymbol{p},\boldsymbol{q},\boldsymbol{v}}(\hat{x})$
2: $\hat{u} = \mathsf{reconstruct}(\hat{u}_b)$
3: $\hat{t} \leftarrow \hat{\Pi}^{\mathsf{Mul}}_n(\hat{x}, \hat{u})$
4: $\hat{m}_b \leftarrow \Pi^{\mathsf{TR}}_{n,n-8}(\hat{t}) \mod 128$
5: $\hat{p}_b \leftarrow \mathsf{extend}_{7,13}(\hat{m}_b) \cdot 2^6 + \hat{e}_b$
6: $\hat{p} = \mathsf{reconstruct}(\hat{p}_b)$
7: **return** $\Pi^{\mathsf{LUT}}_{13,n,\boldsymbol{T}}(\hat{p})$

Figure 11: Protocol for $\mathsf{RecSqrt}_{n,f^{\mathsf{in}},f^{\mathsf{out}}}$

construct the required 13-bit look-up table. Let $\boldsymbol{T} \in \mathbb{U}_N^{2^{13}}$ be a table such that for all $i \in \mathbb{U}_{2^{13}}$, $i = m||e$ where $m \in \mathbb{U}_{128}$ and $e \in \mathbb{U}_{64}$, we have:

$$q = 2^e \cdot (1 + \frac{m}{128}), T[i] \quad = \left\lfloor \sqrt{2^{f^{\mathsf{in}}}/q} \cdot 2^{f^{\mathsf{out}}} \right\rfloor \mod N$$

Based on the above discussion and using the table $\boldsymbol{T}$, we describe the protocol $\Pi^{\mathsf{RecSqrt}}_{n,f^{\mathsf{in}},f^{\mathsf{out}}}$ in Figure 11.

G.3 Overall Protocol for Layer Normalization

G.3.1 Naïve Protocol. A protocol for layer normalization for fixed-point numbers can be implemented as follows. We first locally add the elements of the vector $\boldsymbol{x}$, locally multiply the result with $\left\lfloor 2^f/k \right\rfloor$ and truncate to get m. Then, we locally subtract m from each element in $\boldsymbol{x}$ to get $\boldsymbol{z}$. We then use a beaver-like protocol to compute the sum of squares of the elements in $\boldsymbol{z}$ and call it s. Note that s has precision $2f$. Hence, we truncate by f. Next, we locally multiply the result with $\left\lfloor 2^f/k \right\rfloor$ and again truncate by f to get the variance v. We then use the protocol $\Pi^{\mathsf{RecSqrt}}_{n,f,f}$ (Section 5.3.1) to calculate the fixed-point number corresponding to the reciprocal square root of v, which we securely multiply with each element of $\boldsymbol{z}$ followed by truncation. Finally, we multiply the result with γ, truncate and locally add β.

G.3.2 Optimization. As s is truncated and divided by k before eventually being passed to $\Pi^{\mathsf{RecSqrt}}_{n,f,f}$, we can avoid the truncation and division by k in the protocol by setting $f^{\mathsf{in}} = 2f + \log_2(k)$ while invoking the reciprocal square root protocol. Note that even though fixed-point precision is an integer, here we can use real valued precision as the protocol $\mathsf{RecSqrt}_{n,f^{\mathsf{in}},f^{\mathsf{out}}}$ doesn't impose any restriction on the input precision f^{in} and it is only handled while computing the entries of the LUT.

Based on the above discussion, we provide the protocol $\Pi^{\mathsf{LayerNorm}}_{n,k,f}$ for layer normalization in Figure 12. To avoid invoking reciprocal square root on 0 we add 1 to s in line 6. Here, we note that as the elements of $\boldsymbol{p}$ have an absolute value less than $\sqrt{k}$ (with precision $2f$), leading to a gap, we can use $\Pi^{\mathsf{GapARS}}_{n,f}$ to perform this truncation cheaply. Similarly, as the model weight γ is a number with small magnitude and multiplication with elements of $\boldsymbol{q}$ (bounded by $\sqrt{k}$ in precision f) results in elements with a gap, $\Pi^{\mathsf{GapARS}}_{n,f}$ can be used to truncate vector $\boldsymbol{u}$ as well.

Layer Normalization $\Pi^{\mathsf{LayerNorm}}_{n,k,f}(\hat{\boldsymbol{x}}, \hat{\gamma}, \hat{\beta})$

1: $\hat{y} = \left\lfloor 2^f/k \right\rfloor \cdot \sum_{i=0}^{k-1} \hat{x}[i]$
2: $\hat{m} \leftarrow \hat{\Pi}^{\mathsf{ARS}}_{n,f}(\hat{y})$
3: $\hat{\boldsymbol{z}} = \hat{\boldsymbol{x}} - \hat{m}$
4: $\hat{s}_b = \sum_{i=0}^{k-1} \Pi^{\mathsf{Mul}}_n(\hat{z}[i], \hat{z}[i])$
5: $\hat{s} = \mathsf{reconstruct}(\hat{s}_b)$
6: $\hat{t} \leftarrow \hat{\Pi}^{\mathsf{RecSqrt}}_{n,2f+\log_2(k),f}(\hat{s}+1)$
7: $\hat{\boldsymbol{p}} \leftarrow \hat{\Pi}^{\mathsf{Mul}}_n(\hat{\boldsymbol{z}}, \hat{t})$
8: $\hat{\boldsymbol{q}} \leftarrow \hat{\Pi}^{\mathsf{GapARS}}_{n,f}(\hat{\boldsymbol{p}})$
9: $\hat{\boldsymbol{u}} \leftarrow \hat{\Pi}^{\mathsf{Mul}}_n(\hat{\boldsymbol{q}}, \hat{\gamma})$
10: $\hat{v}_b \leftarrow \Pi^{\mathsf{GapARS}}_{n,f}(\hat{\boldsymbol{u}}) + b \cdot \hat{\beta}$
11: **return** $\hat{v}_b$

Figure 12: Protocol for $\mathsf{LayerNorm}_{n,k,f}$

H SAMPLE SYTORCH CODE

```
TransformerBlock(u64 n_heads, u64 n_embd)
{
    attn = new MultiHeadAttention<T>(n_heads,
                                     n_embd);
    ffn = new FFN<T>(n_embd, 4*n_embd);
    ln0 = new LayerNorm<T>(n_embd);
    ln1 = new LayerNorm<T>(n_embd);
}

Tensor<T> &_forward(Tensor<T> &input)
{
    auto &ln0_out = ln0->forward(input);
    auto &attn_out = attn->forward(ln0_out);
    auto &attn_ip = add(attn_out, input);
    auto &ln1_out = ln1->forward(attn_ip);
    auto &ffn_out = ffn->forward(ln1_out);
    auto &ffn_out_add = add(ffn_out, attn_ip);
    return ffn_out_add;
}
```

Figure 13: SyTorch code for a GPT-2 Transformer block.

I INFERENCE IN THE WAN SETTING

We compare Sigma and CrypTen in the WAN setting in Table 5. Our WAN has bandwidth 293 Mbits per second and ping latency 60ms. The time for secure inference in a WAN is dominated by the time required for communication, and we see that Sigma's FSS-based protocols with low communication overhead allow it to beat CrypTen by $9 - 13\times$.

Table 6: Secure inference of GPT2 with Sigma and CrypTen with varying sequence length.

Sequence length	Time (s)		Comm (GB)	
	CrypTen	Sigma	CrypTen	Sigma
64	14.22	0.96	3.92	0.37
128	20.45	1.61	8.34	0.82
256	36.68	3.26	21.11	1.98
512	85.75	8.01	63.73	5.29
1024	269.06	23.17	228.97	15.92

Table 5: Sigma vs CrypTen on end-to-end inference in the WAN setting. "-" denotes GPU memory overflow.

Model	Time (min)			Communication (GB)	
	CrypTen	Sigma	Speedup	CrypTen	Sigma
BERT-tiny	1.19	0.13	9.1×	0.20	0.02
BERT-base	9.86	0.97	10.1×	8.34	0.99
BERT-large	22.29	2.14	10.4×	23.36	2.63
GPT-2	9.85	0.93	10.6×	8.34	0.82
GPT-Neo	35.07	2.60	13.5×	46.89	4.02
Llama2-7B	-	6.06	-	-	12.35
Llama2-13B	-	8.78	-	-	19.33

J COMPARISON WITH MPCFORMER

MPCFormer [49] replaces cryptographically expensive non-linearities like GeLU with simple quadratic functions and retrains the resulting (simpler) model. Comparing Sigma's performance to MPCFormer is unfair to Sigma, which makes no changes to the underlying plaintext models or their weights. Moreover, MPCFormer uses CrypTen as its backend and if we were to write our models as they are using MPCFormer, it would be the same as writing them in CrypTen. Hence, using MPCFormer as a baseline would be the same as using CrypTen.

K SEQUENCE LENGTH

We evaluate Sigma on input token sequences of lengths between 64 and 1024 in Table 6. For reference, the lengths for inputs in the Lambada dataset are below 180. The speedups of Sigma over CrypTen don't vary much with sequence length. As sequence length increases, the number of GeLUs increases linearly but the compute of softmax increases super-linearly. A sequence length of k requires evaluating k softmax operations with inputs of length k.

Table 8: For different models, we show the size of FSS keys, the time taken by the dealer to generate them, the time to transfer them on the network, and online time of Sigma.

Model	Key size (GB)	Generation time (s)	Transfer time (s)	Online time (s)
BERT-tiny	0.32	0.06	0.27	0.09
BERT-base	16.69	1.43	14.20	1.84
BERT-large	45.06	3.75	38.35	4.73
GPT2	14.17	1.26	12.06	1.61
GPT-Neo	75.57	6.25	64.32	7.43
Llama2-7B	271.65	21.93	231.19	27.01
Llama2-13B	444.36	35.49	378.18	44.13

L ACCURACY RESULTS

Table 7: For different models and datasets, we show the size of the training set (BERT models need finetuning), the size of validation set on which accuracy is measured, the accuracy of PyTorch floating-point, Sigma's accuracy, and the bitwidth BW used by Sigma to get this accuracy.

Model	Dataset	Train Size	Val Size	PyTorch Acc	Sigma Acc	BW
BERT-tiny	SST2	67k	872	81.19%	81.42%	37
	MRPC	3.7k	408	72.54%	72.79%	37
	QNLI	105K	5463	81.64%	81.73%	37
BERT-base	SST2	67k	872	90.59%	90.25%	50
	MRPC	3.7k	408	84.31%	83.82%	50
	QNLI	105K	5463	88.72%	89.03%	50
BERT-large	SST2	67k	872	88.99%	88.99%	50
	MRPC	3.7k	408	78.67%	78.92%	50
	QNLI	105K	5463	92.23%	92.31%	50
GPT2	Lambada	-	5153	32.46%	33.28%	50
GPT-Neo	Lambada	-	5153	57.46%	57.81%	51
Llama2-7B	Lambada	-	5153	70.17%	70.01%	48
Llama2-13B	Lambada	-	5153	73.14%	72.98%	48

M PREPROCESSING COST

We use a dealer to generate FSS keys and transfer them to the machines performing secure inference. Since the dealer has been accelerated with GPUs, the time to generate the keys is small (even smaller than the secure inference time) and the bulk of the preprocessing time goes in transferring the keys from the dealer machines (Table 8). Note that CPU key size is roughly 1.25× larger than the GPU key size for the models in Table 8, due to differences such as the protocols for GeLU (Section 5.1).