Goten: GPU-Outsourcing Trusted Execution of Neural Network Training

Lucien K. L. Ng,[1] **Sherman S. M. Chow,**[1] **Anna P. Y. Woo,**[1] **Donald P. H. Wong,**[1] **Yongjun Zhao**[2]

[1] Department of Information Engineering, The Chinese University of Hong Kong, Hong Kong

[2] Strategic Centre for Research in Privacy-Preserving Technologies & Systems, Nanyang Technological University, Singapore

{luciengkl, sherman, wpy018, wph019}@ie.cuhk.edu.hk, yongjun.zhao@ntu.edu.sg

Abstract

Deep learning unlocks applications with societal impacts, *e.g.*, detecting child exploitation imagery and genomic analysis of rare diseases. Deployment, however, needs compliance with stringent privacy regulations. Training algorithms that preserve the privacy of training data are in pressing need.

Purely cryptographic approaches can protect privacy, but they are still costly, even when they rely on two or more non-colluding servers. Seemingly-"trivial" operations in plaintext quickly become prohibitively inefficient when a series of them are "crypto-processed," *e.g.*, (dynamic) quantization for ensuring the intermediate values would not overflow.

Slalom, recently proposed by Tramèr and Boneh, is the first solution that leverages both GPU (for efficient batch computation) and a trusted execution environment (TEE) (for minimizing the use of cryptography). Roughly, it works by a lot of pre-computation over *known* and *fixed* weights, and hence it only supports private inference. Five related problems for private training are left unaddressed.

Goten, our privacy-preserving training and prediction framework, tackles all five problems simultaneously via our careful design over the "mismatched" cryptographic and GPU data types (due to the tension between precision and efficiency) and our round-optimal GPU-outsourcing protocol (hence minimizing the communication cost between servers). It 1) stochastically trains a low-bitwidth yet accurate model, 2) supports dynamic quantization (a challenge left by Slalom), 3) minimizes the memory-swapping overhead of the memory-limited TEE and its communication with GPU, 4) crypto-protects the (dynamic) model weight from untrusted GPU, and 5) outperforms a pure-TEE system, even without pre-computation (needed by Slalom). As a baseline, we build CaffeScone that secures Caffe using TEE but not GPU; Goten shows a $6.84\times$ speed-up of the whole VGG-11. Goten also outperforms Falcon proposed by Wagh *et al.*, the latest secure multi-server cryptographic solution, by $132.64\times$ using VGG-11. Lastly, we demonstrate Goten's efficacy in training models for breast cancer diagnosis over sensitive images.

Introduction

Advances in data science are undoubtedly changing our lives. In particular, deep neural networks (DNN) show unprecedented performance in many life-changing applications, such as genomic analysis of rare diseases, medical image analysis (*cf.*, decision-tree classifiers (Tai et al. 2017; Ma et al. 2021)), and child exploitation imagery (CEI) detection (*cf.*, PhotoDNA[1] by Microsoft, which simply compares the hashes of the images) (Wagh et al. 2021). Their success requires voluminous data, but *stringent privacy regulations* are *curbing* data collection. This motivates a flurry of research in privacy-preserving machine-learning algorithms in recent years, such as private inference (Chandran et al. 2019; Wong et al. 2020; Ng and Chow 2021). *Private training* is much more complicated, as indirectly reflected by the fact that recent solutions (*e.g.*, (Mohassel and Rindal 2018)) resort to the *secure multi-server computation model*, requiring the servers (up to four (Chaudhari, Rachuri, and Suresh 2020)) to not collude with each other. This model enables processing private data via lightweight techniques, such as secret sharing, in contrast to single-server approaches (that often use homomorphic encryption). Nevertheless, it is still orders of magnitude slower than plaintext computation. Falcon (Wagh et al. 2021), the state-of-the-art using three servers, takes *weeks* to train a neural network for classifying CIFAR-10, a medium-level image classification dataset.

We note that no existing private training approaches can leverage GPU, albeit being a usual practice in (plaintext) DNN training. Meanwhile, we see a trend in using trusted execution environments (TEEs) to minimize the use of cryptography in processing private data (Shaon et al. 2017; Hynes, Cheng, and Song 2018). We are thus intrigued to ask:

> *Can we support DNN* training *(and prediction) by using TEE and* untrusted *GPU while preserving the privacy of all stakeholders? How much speedup can we gain?*

The security guarantee of TEE is bounded within the CPU and its fixed memory. It is non-trivial to use GPU for processing private data efficiently. Specifically, GPU does not natively support cryptographic operations over a finite field.

Solving Five (Open) Problems in One Scheme

To better understand the challenges in solving the above problems, we revisit how Slalom (Tramèr and Boneh 2019), the state-of-the-art TEE+GPU solution, performs private *prediction*, and why it fails to support private training. The core idea of Slalom can be described in simple terms. Firstly,

[1] https://www.microsoft.com/en-us/photodna

for a linear function f, $f(r)$ is *precomputed* for a one-time random *blinding factor* r in $\mathbb{Z}_q$ (q is a large prime). When the input x is known, it first applies *static* quantization on x (to make it an element of the cryptographic *finite* field $\mathbb{Z}_q$), then outsources $x' = x + r \bmod q$ to the *untrusted* GPU for computing $f(x + r)$. Given x', there always exists r for each possible x, hence providing secrecy (as one-time pad). When TEE gets back $f(x + r) = f(x) + f(r)$ (f is linear), it "unblinds" using $f(r)$ to obtain $f(x)$, the private result.

Five challenges remain unsolved for training.

1. Slalom critically relies on a lot of precomputation. Each input x consumes one such $f(r)$, and asking the untrusted GPU to compute $f(r)$ is insecure (no protection over $f(x)$). If we just ask the TEE to compute $f(r)$, it is of the same complexity as $f(x)$. If we load them to the TEE on-spot, they are subjected to the memory limit (practically $\sim$90MB) and incur unwanted communication overhead. Minimizing loadings to TEE is one of our design goals.
2. Such precomputation only works when W is a *fixed* parameter (of $f_W(x) := x \cdot W$), which is naturally changing during training, making precomputed $f_W(r)$ useless.
3. Slalom works since the untrusted GPU knows W of f_W, but it should be protected in private training (or securely-outsourced prediction) – a challenge explicitly left open.
4. Slalom works with a fixed q and explicitly left *dynamic quantization* as one of the open challenges. Also, as mentioned, DNN weights are not fixed in training. Weight fluctuations further complicate dynamic quantization.
5. Finally, Slalom is an offline/online design but is not a "truly" outsourcing solution. The time needed for the TEE (*i.e.*, computing $f(r)$) is the same as the time for the job to be outsourced (*i.e.*, computing $f(x)$). Once the precomputation is "used up," no more outsourcing is possible.

GPU-Outsourcing Trusted Execution of NN

We propose Goten, the first GPU+TEE framework that protects prediction queries, *training data, and model parameters*. Goten achieves a higher throughput without worrying that the offline preparation will be exhausted when the demand reaches its peak. We thus achieve "true" outsourcing – the time needed for securely outsourcing the job to the untrusted GPU is less than that for computing the job locally by the TEE *plus any time needed for precomputation*.

Empirical Evaluation. This is also the first work with extensive experimental investigations of the above possibility. Concretely, we show that we can improve the performance of VGG11 by $6.84\times$, achieving the highest efficiency so far. Our code is available at github.com/goten-team/Goten.

Dynamic Quantization Scheme. We quantize the neural network parameters to fixed-point number format for efficient cryptographic operations (*cf.*, *static* quantization in Slalom). This process needs to be implemented carefully:

1. Matrix multiplications in DNN scale up the outputs quickly. The data type's numeric limit is easily exceeded.
2. Some functions map values to a small interval (*e.g.*, $\mathrm{softmax}()$, $\mathrm{sigmoid}()$), which require high precision.

To avoid these potential accuracy problems, we developed a *data-type conversion scheme* for enjoying "the best of both worlds," *i.e.*, the benefit of accurate floating-point operations on TEE and efficient fixed-point operations on GPUs. Our experiment confirms Goten's high accuracy.

Secure GPU-Outsourcing Protocol. Finally, we design a new outsourcing protocol (from TEE to the untrusted GPU) that speeds up linear computations (*e.g.*, matrix multiplication). It prevents leaking information to the hosts of (untrusted) GPU via additive secret sharing. While secret sharing has been extensively used, existing designs assume a general scenario and do not consider the characteristics of TEE and GPU. Our protocol leverages *the best of TEE* (for deriving randomness) *and GPU* (for batch processing).

Also, the CPU needs to convert data of linear layers into the format used by secret sharing and then convert the secret-shared results from GPU back into the usual format for non-linear layers. We call these procedures *pre-processing* and *post-processing* of outsourcing. We need a clever design to make sure their cost will not offset the performance gained.

A trivial approach to protecting two operands a and b is to encrypt them to the TEE enclave and ask it to multiply them directly. However, it cannot enjoy the batch-processing advantage of GPU and is inefficient for large-scale computation. Our protocol uses the enclave to secure the unprotected computation environment of GPU, without the enclave performing any relatively expensive decryption beyond the "bare minimum," *i.e.*, two instances of decryption (for the two operands). Our protocol is also round-optimal. Once the enclaves are "bootstrapped," they only need to outsource the computation with one message flow and then get back the results from the GPUs with another. It is important for minimizing the communication overhead between the servers (while TEE and GPU are co-located in Slalom). We achieve this by our adaption of the original Beaver's protocol (Beaver 1991) for multiplications of additive shares.

Memory-aware Measures. We utilize Intel SGX as TEE, which has a memory limit of 90MB (Shaon et al. 2017) among the 128MB limit claimed by Intel. It is not sufficient for training large neural networks like VGG. Although paging in Linux SGX SDK can over-subscribe memory, it imposes much performance overhead ($10\times$ to $1000\times$) over unprotected programs (Arnautov et al. 2016) for exiting the enclave mode and switching back after processing the untrusted memory. To prevent the overhead that might nullify Goten's performance gain, we take extra measures to reduce overheads by looking into our specific DNN operations and using the enclave itself to handle memory swapping.

Summary of Contribution. As the pipeline in Figure 1, Goten *dynamically* (de-)quantizes the inputs (and outputs) of linear layers according to SWALP (Yang et al. 2019), a low-precision training scheme, to cater for the finite field used by our new secure GPU outsourcing protocol. Our protocol enjoys (untrusted) GPU's high performance with protection but without sacrificing accuracy or incurring high costs communicating data to or processing data within the TEE that may nullify any performance gain from GPU outsourcing.

Preliminaries

(A Simplified View of) Neural Networks

Linear layers have a basic form of $y = w \otimes x$, where x is the input, and w is the learnable parameters we aim to protect. $\otimes$ is usually matrix multiplication, convolution, or their corresponding operations for computing gradient, *e.g.*, transposed convolution. Linear transformation is the most computationally intensive part (Jia 2014). It remains so no matter in plaintext or in Goten (as shown in our experiment).

For non-linear layers, we have —
i) *Activation layer* applying non-linear functions on its input.
ii) *Pooling layer* applying aggregation, *e.g.*, $\max()$, $\mathrm{mean}()$.
iii) *Normalization layer* normalizing each pixel by subtracting the mean and dividing by the variance (*e.g.*, batch normalization deriving statistics parameters along the batch).

Very deep convolutional network (VGG) is a family of very deep DNN with 9 to 19 layers with parameters and has extraordinary performances on visual tasks. Supporting VGG for private training/inference is thus desirable.

Two-Party Computation via Secret Sharing

Servers U_0 and U_1 holding $a, b \in \mathbb{Z}_q$ respectively can let a third server learn $c = a + b$ without revealing a, b as follows.

1. U_0 picks $\langle a \rangle_0 \in \mathbb{Z}_q$ uniformly at random.
2. U_0 then sends $\langle a \rangle_1 = a - \langle a \rangle_0$ to U_1.
3. U_1 also uniformly samples $\langle b \rangle_0 \in \mathbb{Z}_q$ and sends it to U_0.
4. U_1 keeps $\langle b \rangle_1 = b - \langle b \rangle_0$ privately.
5. U_0 computes $\langle c \rangle_0 = \langle a \rangle_0 + \langle b \rangle_0$.
6. U_1 computes $\langle c \rangle_1 = \langle a \rangle_1 + \langle b \rangle_1$.

U_0 and U_1 now hold shares of $c = \langle c \rangle_0 + \langle c \rangle_1 = a + b$.

Useful Subroutines: $\mathrm{Rand}(r_x)$ is a *pseudorandom function* that takes a *random seed* r_x and outputs a random $x' \in \mathbb{Z}_q$ for one-time-padding x. We also define $\mathrm{Gen}_i(r_x, x) := (-1)^i \cdot \mathrm{Rand}(r_x) + i \cdot x$ for $i \in \{0, 1\}$. We let U_i for $i \in \{0, 1\}$ hold either $\mathrm{Rand}(r_x)$ or $x - \mathrm{Rand}(r_x)$ as a share of x.

Beaver's protocol (Beaver 1991) lets U_0 with $\langle a \rangle_0, \langle b \rangle_0$ and U_1 with $\langle a \rangle_1, \langle b \rangle_1$ compute secret shares of $c = ab$.

1. We suppose U_0 and U_1 have precomputed in an offline stage *additive* secret shares of u, v, and z where $u \cdot v = z$, *i.e.*, U_i has $(\langle u \rangle_i, \langle v \rangle_i, \langle z \rangle_i)$ as a *Beaver's triplet*.
2. U_i computes $\langle e \rangle_i = \langle a \rangle_i - \langle u \rangle_i$ and $\langle f \rangle_i = \langle b \rangle_i - \langle v \rangle_i$.
3. They then exchange $\langle e \rangle_i$ and $\langle f \rangle_i$ to reconstruct e and f, masking a and b respectively: $e = a - u$ and $f = b - v$.
4. Finally, with e and f, they compute $\langle c \rangle_i = -i(e \cdot f) + f \cdot \langle a \rangle_i + e \cdot \langle b \rangle_i + \langle z \rangle_i$ locally, where $\langle c \rangle_0 + \langle c \rangle_1 = ab$.

Using this protocol as-is requires *two rounds* of communication (for recovering (e, f)) and precomputation (of shares of (u, v, z)). Also, the input/output is in secret shares, while we can afford to let the enclaves (securely) store the input/output directly. Our protocol aims to reduce communication and precomputation costs, improving the throughput.

The Design of Goten

The System Setting and Assumptions

Goten uses three *non-colluding* servers S_0, S_1, and S_2. They interact for privacy-preserving computation conforming to the specification of our protocol (as the code integrity can be ensured by TEE). They hold secret shares of the training data, model parameters, and intermediate values. The values hidden by the secret shares are not recoverable by any single one of them. These non-colluding servers can be completing cloud service providers or prestigious institutions. If we want to only use 2 servers, we can replace S_2 with an *offline* preparation phase (see the description of our protocol).

Servers S_0 and S_1 are equipped with TEE-enabled CPUs (*e.g.*, Intel SGX) to instantiate trusted enclaves E_0 and E_1, respectively. In our protocol (Figure 2), we use U to denote an untrusted GPU of a server, which interacts with the TEEs.

Overall Workflow

Goten uses enclaves to perform most of the computation, except for linear layers. Enclaves guarantee the correctness and confidentiality of the computation, but their performance is worse than GPU's, especially for (linear) matrix multiplication and convolution. We thus devise a secure protocol that outsources the linear layers to GPU for efficiency.

Once the data goes out of the enclaves, it is in an *untrusted zone*. The host can read it, and the TEEs no longer ensures its privacy. So we need to take extra steps to protect the data dispatched to GPUs. The extra data transfer between servers and between CPU and GPU may slow down the outsourcing process. Our protocol needs to minimize these costs.

We focus on explaining our training protocol. As in Figure 1, we have two enclaves E_0 and E_1, residing in S_0 and S_1. During initialization, the enclaves synchronize to make sure their inputs, in particular, the training data, DNN configurations (*e.g.*, its architecture and hyper-parameters), and randomness (derived via seeds r_u, r_v, r_a, r_b, r_z in Figure 2, which can be easily established by a key-exchange protocol) are the same. The training data providers attest both enclaves (a basic TEE feature) for having loaded in the right program code and specification. Upon confirming their integrity, the data provider establishes secure channels with them and sends training data to either one of the enclaves.

Both enclaves run the (stochastic) gradient descent training algorithm. They sample the same training batch, take almost identical steps (except in linear layers) for forward and backward propagation, update the model parameters according to the gradient, and repeat these steps until meeting the pre-defined stop criteria. Both enclaves run alike steps with the same randomness, so many of their "helper values" are the same. This trick helps minimize the communication cost and round of our GPU-outsourcing protocol.

For inference, Goten simply performs a forward propagation and returns the output layer's result to the querier, without the backward propagation and repeated iterations.

Secure GPU-Outsourcing Protocol

A linear layer basically performs linear operation $\otimes$ on two tensors a and b, and outputs the resulting tensor $c = a \otimes b$.

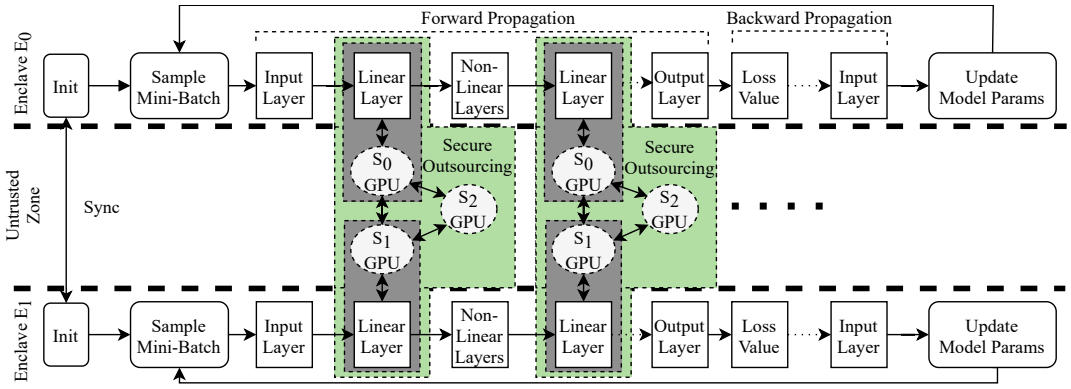

Figure 1: Overview of Goten: Crypto-Aware Private Training by Servers S_0, S_1, and S_2 using Secure GPU-Outsourcing Protocol

Our Secure GPU-Outsourcing Protocol for $c = a \otimes b$

1 : $\mathsf{U}_2 : u \leftarrow \text{Rand}(r_u), v \leftarrow \text{Rand}(r_v)$
2 : $\mathsf{U}_2 \rightarrow \mathsf{E}_0, \mathsf{E}_1 : z = u \otimes v$
3 : **for** $i = 0, 1$ (in parallel)
4 : $\quad \mathsf{E}_i \rightarrow \mathsf{U}_i : \langle a \rangle_i \leftarrow \text{Gen}_i(r_a, a), \langle b \rangle_i \leftarrow \text{Gen}_i(r_b, b),$
5 : $\quad\quad e = a - \text{Rand}(r_u), f = b - \text{Rand}(r_v)$
6 : $\quad \mathsf{U}_i \rightarrow \mathsf{E}_0, \mathsf{E}_1 : c_i = \langle a \rangle_i \otimes f + e \otimes \langle b \rangle_i - i \cdot e \otimes f$
7 : **endfor**
8 : $\mathsf{E}_0, \mathsf{E}_1 : c = z + c_0 + c_1$

*$\{r_a, r_b, r_u, r_v\}$ are the pre-agreed random seeds of E_0, E_1

Figure 2: Outsourcing Computation of $a \otimes b$ to the GPUs

TEEs $\mathsf{E}_0, \mathsf{E}_1$ *resided* in servers S_0 and S_1 outsource a and b to the GPUs within S_0 and S_1 for computing $c = a \otimes b$ with better performance. Different from the typical two/multi-server computation model, which processes inputs in share $(\langle a \rangle_i, \langle b \rangle_i)$ and ends up with $\langle c \rangle_i$, we load (a, b) as-is to the enclaves, which obtain c as-is as the result.

Figure 2 describes our protocol for outsourcing linear operation of $a \otimes b$. GPU U_2 of S_2 generates a random Beaver triplet (u, v, z), where $z = u \otimes v$, and sends z to E_0 and E_1. All instances of "$\rightarrow \mathsf{E}_i : var$" in Figure 2 refer to loading the variable(s) var to E_i. This explains Line 2 (and Line 6).

E_0 and E_1 then create additive shares of a and b, and their masked values e (a masked by u) and f (b masked by v). U_0 gets $(\langle a \rangle_0, \langle b \rangle_0, e, f)$ and U_1 gets $(\langle a \rangle_1, \langle b \rangle_1, e, f)$. After the GPUs locally carried out linear computations over these values, the results are returned to E_0 and E_1 (Line 6) for them to recover the computation result by also using z.

Correctness. Our protocol reconstructs c by $z + c_0 + c_1$ at the last line of Figure 2. By $z + c_0 + c_1 = u \otimes v + \sum_{i=0}^{1} \langle a \rangle_i \otimes f + e \otimes \langle b \rangle_i - i \cdot e \otimes f = u \otimes v + a \otimes (b - v) + (a - u) \otimes b - (a - u) \otimes (b - v)$ and the linearity of $\otimes$, we have $c = a \otimes b$.

Reduced Computation and Communication Cost. In the original Beaver's protocol, the shares $(\langle a \rangle_0, \langle b \rangle_0)$ and $(\langle a \rangle_1, \langle b \rangle_1)$ from parties U_0 and U_1 must be masked independently by the corresponding one-time pads $(\langle u \rangle_0, \langle v \rangle_0)$ and $(\langle u \rangle_1, \langle v \rangle_1)$. They also need to interact to recover (e, f).

Our protocol aims to compute $a \otimes b$ by operating over (e, f), a masked version of (a, b). Enclaves E_0 and E_1 use the same seeds (r_u and r_v) to derive u and v. Both of them know a and b, so they can obtain e and f without any interaction. This saves half of the pre/post-processing and communication costs and makes e and f no longer dependent on $\langle a \rangle_i$ and $\langle b \rangle_i$. E_0 and E_1 thus can run the steps in Figure 2 in parallel. Also, E_0 and E_1 no longer need to interact until they reconstruct the result c. We then further reduce the run-time of such pre-processing to roughly $1/4$ of the original.

Table 1 compares the communication cost of the original Beaver's protocol and ours. In both, the task of the third server U_2 is to provide the secret shares of u, v, z to U_0 and U_1. Let s_a, s_b, and s_c be the size of a, b, and c. Our protocol reduces communication costs by roughly 75% and reduces the round of communication to 2.

Larger Batch Size for Higher Training Throughput. When the batch size increases, the number of inputs in each layer also increases. More data can be dispatched to the GPU or other servers at the same time to amortize the communication latency. Meanwhile, a large batch size can better utilize GPU's batch processing power, leading to a higher GPU/CPU speedup ratio. Nevertheless, if the batch size is too large, the intermediate elements may exceed the memory limit of GPU. For the best performance, one should pick the largest possible batch size within such a limit.

Removing S_2. Apart from "fully" utilizing the enclaves to do what they are good for, we also choose to "fully" leverage the non-colluding assumption (needed by the original protocol (Beaver 1991)) with one more server S_2 to establish the triplets $\{\langle u \rangle_i, \langle v \rangle_i, \langle z \rangle_i\}$ where $u \otimes v = z$. If one wants to remove S_2, the preparation of (u, v, z) can be done by E_0 and E_1 themselves or a group of triplet providers (not necessarily with GPUs) in an *offline* phase. They can be prepared in parallel as triplets are independent. Our protocol in Figure 2 can be slightly adapted. (Details in the full version.)

(De-)Quantization

Quantization for Secure Computation. Our protocol performs linear operations over fixed-point numbers in $\mathbb{Z}_q$, while common neural networks operate over floating-point

	Server	Beaver's Protocol	Goten
Size	S_0	$3(s_a + s_b) + 2s_c$	$2s_a$
	S_1	$3(s_a + s_b) + 2s_c$	$2s_b$
	S_2	$2(s_a + s_b + s_c)$	$2s_c$
Round	S_0	3	2
	S_1	3	2
	S_2	2	2

Table 1: Communication-Cost Comparison with Beaver's

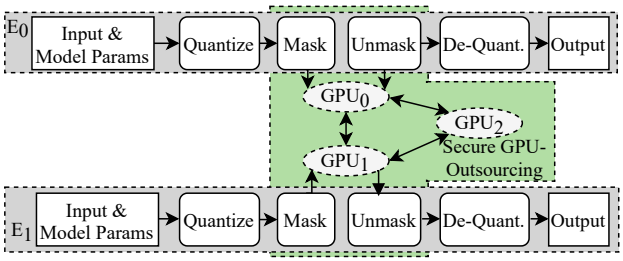

Figure 3: Dynamic (De-)Quantization + Secure Outsourcing

numbers ("floats"). To ensure Goten can train common neural networks, we quantize the inputs of linear layers and dequantize their outputs as in Figure 3. So the fixed-point requirement only affects linear layers but not the rest of neural networks, *e.g.*, non-linear computation or model parameters update with gradient, hence attaining higher accuracy.

Formally, we approximate a floating-point linear operation $x \otimes_{\mathsf{f}} w$ by a fixed-point linear operation $\otimes_{\mathbb{Z}_q}$ that only takes $\mathbb{Z}_q$ elements as inputs (and outputs), and we need a quantization scheme to convert floats to $\mathbb{Z}_q$ and vice versa. We first quantize x and w into $x_Q = Q(x; \theta_x)$ and $w_Q = Q(w; \theta_w)$ with quantization parameters θ_x and θ_w. We then use fixed-point multiplication $\otimes_{\mathbb{Z}_q}$ to compute $y_Q = x_Q \otimes_{\mathbb{Z}_q} w_Q$, and derive the result by $y = Q^{-1}(y_Q; \theta_x, \theta_w) \approx x \otimes_{\mathsf{f}} w$.

The Constraints of GPU. Off-the-shelf optimized libraries for GPU do not support integer modular arithmetic. To work on $\mathbb{Z}_q$ integers, we put them as 64-bit double-precision floats as Slalom (Tramèr and Boneh 2019). This leaves us only 53 significant bits plus a sign bit to represent the integers in linear layers (the rest of $(64 - 53 - 1)$ exponent bits are 0).

Such a *bitwidth* harshly restricts input data's magnitude. To ensure the matrix multiplication or tensor convolution $a \otimes b$ does not overflow, we need $q^2 k < 2^{53}$, where k is the number of addition needed to compute an entry in the resulting tensor, *e.g.*, the number of columns of matrix a in matrix multiplication. We also need to prevent overflow in $\mathbb{Z}_q$ since $\otimes$ not only operates over the secret shares but also the value hidden within. Suppose the magnitude of the input values is bounded by p; we further require $p^2 k < q$. Combining both requirements above, we have $p^4 k^3 < 2^{53}$.

Putting it in hardware terms, each input values theoretically can be of $\sim$13 bits because $\log_2 p < 53/4$ when $k = 1$. However, we need some safety margin for k as it may be up to thousands. In view of such severe constraints, we should pick a quantization scheme that keeps p as little as possible while the quantization error is small enough for training.

Challenges in Dynamic Quantization. In prediction, the model is fixed. Slalom thus knows the value distribution of model parameters for deriving the distributions of the input, output, and intermediate values. Picking a *static* scaling parameter that minimizes the prediction error is thus relatively easy: $Q(\cdot; \theta)$ is always parameterized by $\theta = 2^8$. Slalom states that quantization for training is challenging since the range of gradient of the weight may change, so do the input and output of the successive layers. *Knowing the value distribution prior to training is hard*. We do not know beforehand what the quantization parameters should be.

Beyond what Slalom did, we need *dynamic* quantization for training, meaning that it can adapt the change on the distribution of the model parameters, and hence the intermediate value and gradient. The (de-)quantization process *also* has to be *efficient* to avoid being the bottleneck that may cancel out the performance gain from GPU-outsourcing.

Dynamic Quantization for Training

SWALP (Yang et al. 2019) is a scheme that employs stochastic weight averaging for training in a low-precision (or low-bitwidth) setting. The forward and backward computations of linear layers are performed in low-bitwidth fixed-point, but the weights are stored and updated in floats with high-bitwidth. Let bit be the number of bits available for low-bitwidth computations (defaults to 8). For input and weight, SWALP finds out the maximum absolute value and calculates its exponent in the format of bits, *i.e.*, compute $\mathsf{exp} = \lfloor (\log_2 \circ \max \circ \mathrm{abs})(\mathsf{data}) \rfloor$. Then, it rescales all the values by 2^{exp} so that the new maximum values are roughly aligned to $2^{\mathsf{bit}-1}$, rounds them up stochastically (Gupta et al. 2015), and clips all the values to $[-2^{\mathsf{bit}-1}, 2^{\mathsf{bit}-1}]$, *i.e.*, $\mathsf{data}_Q = Q(\mathsf{data}, \mathsf{exp}) = \mathsf{clip}(\lfloor \mathsf{data} \cdot 2^{-\mathsf{exp}+\mathsf{bit}-2} \rceil)$, where $\mathsf{clip}(a) = \min(\max(a, -2^{\mathsf{bit}-1}), 2^{\mathsf{bit}-1})$. After computation, the results are scaled-down via $y = y_Q \cdot 2^{\mathsf{exp}_x + \mathsf{exp}_w - 2 \cdot \mathsf{bit} + 4}$.

SWALP dynamically quantizes depending the sampled maximum values of the weight and input in every iteration. Finding the maximum absolute value and scaling up and down the values requires only 3 linear scans. The scaling can be fused with other pre/post-processing too. From the existing SWALP experiment, its accuracy drops by less than 1pp when compared to training in full-precision for VGG11, and the operands are only 8 bits. By adopting SWALP, Goten can train VGG11 with high accuracy as we will see.

Our Memory-aware Mechanism

Computations in linear layers and any necessary pre/post-processing could suffer from paging overheads. As reported by SCONE[2], memory access can be $10-1000\times$ slower than the plaintext setting. Eleos (Orenbach et al. 2017) explains that triggering native paging would exit the enclave mode, which is time-consuming. Cosmix (Orenbach et al. 2019) can mitigate these issues, but integrating it with a DNN framework is non-trivial. Our memory-aware measures let

[2]SCONE is a Secure CONtainer Environment (Arnautov et al. 2016) that allows developers to directly run applications in an SGX enclave with almost no code change. TensorSCONE (Kunkel et al. 2019), which is not open source, used it to run TensorFlow.

the enclave specifies the piece of memory to be used and handles most operations in the enclave to minimize paging.

When Goten needs to allocate more memory than SGX's memory limit, it stays in the enclave mode, directly encrypts the chunk of memory, and evicts it to the untrusted zone. When it needs the data outside the enclave, it loads the chunk of memory into the enclave for decryption. For operations in the enclave, we aim to minimize the memory access across the border of the trusted/untrusted zone. In particular, we fuse together operations that use the same set of memory, and independently handle batches in non-linear layers.

Empirical Evaluation

For Goten, its SGX part is written in C++ and compiled with Intel SGX SDK 2.6. All C++ code is compiled by GCC 7.5, with flag `-march=native` to avoid data-dependent branching. We use PyTorch 1.2 on Python 3.6.9 to marshal network communication and operation on GPU, which runs with CUDA 9.0. We reuse some code of Slalom (Tramèr and Boneh 2019), including their code of cryptographically-secure random number generation and encryption/decryption, and their OS-call-free version of the linear algebra library Eigen. All experiments were conducted at least 5 times, and we report the average result. The deviations of our running times from the average are $<10\%$.

We record the timing figures for network communication and GPU computation separately on 3 Google Cloud VMs, each equipped with 32GB RAM and an Nvidia V100 GPU. These machines can communicate at 8Gbps with less than 5ms latency. Unfortunately, all CPUs on Google VMs do not support SGX hardware mode. We thus run Goten in the simulation mode (which skips the data protection). This suffices for timing communication and GPU computation.

For the timing figures affected by the simulation mode, they are replaced by the figures produced from our desktop computer using the SGX hardware mode (with all the protection of SGX). The desktop is equipped with Intel i7-7700 Kaby Lake Quad-cores 4.3GHz CPU and 16GB RAM, using Ubuntu 18.04. We use it to run all non-linear layers and the pre/post-processing for linear layers (generating additive masks, recovering the secret, and moving data to/from the unprotected memory zone). These are all operations affected by the difference between hardware and simulation modes. We also evaluate our baseline without GPU on this machine.

Our Baseline: CaffeeScone. We combine SCONE with Caffe (Jia et al. 2014), an open-source DNN framework, to build our baseline privacy-preserving DNN framework – CaffeScone. CaffeScone does not use GPU or non-colluding servers. Beyond showing what one can get by applying a generic solution (SCONE) that uses SGX for training (not supported by Slalom (Tramèr and Boneh 2019)), our CaffeScone implementation enables more benchmarking for insight in improvements, which are eventually achieved by our main result (hence further optimizing it is not our goal).

Experiment Overview. We evaluate Goten and CaffeScone on CIFAR-10 (Krizhevsky and Hinton 2009), an image classification dataset commonly used for accuracy benchmark (Tramèr and Boneh 2019; Ng and Chow 2021). It contains 60000 32×32 3-color-channel images divided into 10 classes, 50000 of them are for training, and the rest is for testing. Good performance on this benchmark means that the neural networks are likely to work also well on other visual applications, *e.g.*, prohibitive image detection.

Framework	GPU / TEE	TP	Speedup
Falcon	✗/ ✗	1482	$132.64\times$
CaffeScone	✗/ ✓	28800	$6.84\times$
Goten	✓/ ✓	196733	-

Table 2: Training Throughput (TP: images/hr) on CIFAR-10

We pick VGG architecture with 11 layers and batch normalization layers as a typical DNN that can attain a high accuracy on CIFAR-10 but small enough to fit with (the memory limit of) CaffeScone. For Goten, we also use VGG-11.

Our experiments aim to answer the following questions:

- Can Goten beat the state-of-the-art training framework?
- What is the training throughput of Goten? How much do we gain? Goten processes linear and non-linear layers differently. What are the corresponding performance gains?
- Since adopting SWALP may change the training convergence speed, by how much the GPU-outsourcing protocol and SWALP as a whole can improve the training efficiency for attaining a particular testing accuracy?
- What is the performance of Goten for sensitive tasks, says, medical diagnosis over images?
- As network conditions between the servers are critical for the training efficiency, how does the training speed vary with the bandwidth? Also, what is the minimum bandwidth for Goten to perform better than CaffeScone?

Comparison with the State of the Art. Table 2 shows our speedup over Falcon (Wagh et al. 2021), which has the prior best performance on training throughput on CIFAR-10 under the LAN setting. Falcon adopts VGG-16, a slightly larger neural network, whose computational cost is at most a double of VGG-11. It did not provide any accuracy figures.

Training Throughput. We compare Goten's training throughput (the batch size divided by the processing time for each batch) with CaffeScone, our baseline approach.

Goten uses both TEE and GPU, while CaffeScone only uses TEE. Naturally, the memory limit would be different. As explained earlier, for maximizing their performance, we pick the largest possible batch size that fits the memory. For our experiment setting of GPU with 16GB, we pick 512 as the batch size for Goten. For CaffeScone, the memory limit of the SGX enclave is only 90MB; it attains its highest throughput with the batch size of 128 and 2 CPU cores (as confirmed by our experiment detailed in the full version).

Table 3 shows the *speedup* of Goten over CaffeScone in terms of training throughput. For linear layers, it is $6.17\times$. For non-linear layers, the speedup is largely due to our memory handling as non-linear computation is not outsourced (and VGG-11 is large enough to trigger the memory paging, done by the Linux system in our experiment), which is $8.02\times$. In total, Goten outplays CaffeScone by $6.84\times$.

Framework (Batch size)	Linear	Non-Linear	Total
CaffeScone (128/batch)	9243	6774	16017
Goten (512/batch)	5990	3378	9368
Speedup on Throughput (×)	6.17	8.02	6.84

Table 3: Time (in ms) and Throughput of Different Layers

Accuracy	85%	86%	87%	88%	89%	90%
Speedup (×)	10.82	6.71	6.71	6.69	4.93	-

Table 4: Accuracy vs. Speedup using Goten over CaffeScone

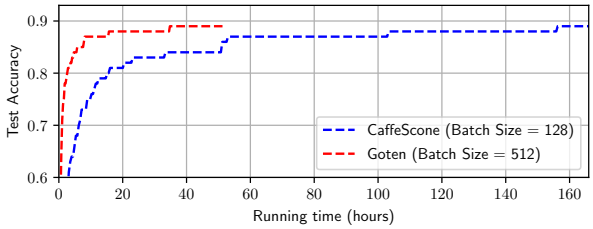

Figure 4: Convergence Time of VGG-11 over CIFAR-10

Accuracy	81%	82%	83%	84%	85%	86%
Speedup (×)	8.53	13.67	4.27	6.33	3.42	7.28
Time (minute)	1.25	1.56	13.1	16.9	31.2	46.8

Table 5: Training Time and Speedup on VGG-11 over IDC

Convergence on Quantized NN. CaffeScone's training is done over single-precision floats, while Goten uses SWALP. They both ran stochastic gradient descent with momentum of 0.9, weight decay of $5{\cdot}10^{-4}$, and learning rate of 0.5. The learning rate was halved every 30 epochs. Goten, with quantization, may take more steps to attain a particular accuracy than CaffeScone, leading to longer training time despite higher training throughput. To dispel the doubt, we record the convergence time of both captured on GPU and rescale the time axis according to Table 3's timing. Figure 4 demonstrates how the speedup leads to a higher convergence rate, by which we confirmed that Goten converges much faster. Also, Goten can attain 89% accuracy within 40 hours.

Table 4 shows that our quantization attains a high accuracy in a shorter time. Notably, Goten reaches 85% accuracy 10.8× faster than CaffeScone. However, Goten cannot attain 90% accuracy within 200 epochs, but CaffeScone can.

Sensitive Training Tasks. To showcase Goten's ability in deep learning over sensitive data, we train with VGG-11 on a public dataset (Cruz-Roa et al. 2014) consists of images of women's breast tissue, which can be used to detect invasive ductal carcinoma (IDC), the most common type of breast cancer. We trained over pre-processed (Janowczyk and Madabhushi 2016) images. As shown in Table 5, Goten attains 82% accuracy in 94s, 13.6× faster than training using CaffeScone. For 86% accuracy, Goten takes 46.8 mins, 6.4× faster than CaffeScone. Notably, the original works (Janowczyk and Madabhushi 2016) attained only 84% accuracy with AlexNet, which is smaller than VGG-11.

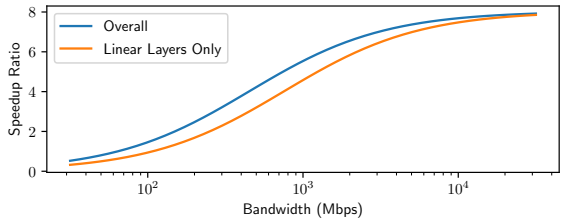

Figure 5: Speedup Ratio vs. Bandwidth (megabit/second)

Speedup Ratio in Different Network Settings. The servers need to send the shares of resulting tensors to other servers for the enclaves to reconstruct the results. To investigate Goten's performance under various network settings and figure out the minimum bandwidth for performance gain over CaffeScone, we record the runtime of each operation in linear layers, single out the communication delay, and recalculate it with respect to other bandwidth settings.

Figure 5 shows the speedup ratio according to different bandwidth with a fixed latency[3] of 20ms. We skip the speedup ratio of non-linear layers because it is independent of the network performance. To prevent performance degradation, the minimum bandwidth is 100Mbps; and the performance gain saturates around 10Gbps for 8× speedup. We expect that companies joining forces to perform training are motivated to dedicate a better network line between them.

Using only 2 Servers. As described earlier, We can remove S_2 by shifting its preparation job to the other two enclaves during the offline phase. We estimate the offline throughput by recording the time of running the linear layers solely in CaffeScone. Each of E_0 and E_1 spawns 2 CaffeScone instances, totaling 4 instances. We pick a batch size of 512 as used by Goten and assign each instance 2 threads.

Albeit the offline computation relies solely on CPU, parallelization on enclaves can cancel the disadvantage. Moreover, the tasks in S_2 are less intensive. In the end, the offline throughput is just slightly higher (1.03×) than the online throughput. The training parties can thus avoid using the third server at the price of doubling the runtime.

Conclusion

We proposed Goten, with a secure outsourcing protocol leveraging the best of TEE and GPU, memory-aware measures to mitigate the paging overheads, and careful treatments in data type to ensure efficiency yet avoiding overflow.

Goten uses a dynamic quantization scheme to cater to the fluctuation in the weight during training, and it significantly outperforms the state-of-the-art purely cryptographic approaches. Devising cryptographic protocols that leverage GPU is an ongoing research topic (Ng and Chow 2021). We would like to call for (collaboration in) future works towards making GPU-enhanced approaches more widely applicable, *e.g.*, by experimenting with "real-world" sensitive data.

[3]The latency of geometrically-close US servers of Google VM (*e.g.*, ∼12ms between us-east1 and us-east4) can be found at https://docs.aviatrix.com/HowTos/gcp_inter_region_latency.html

Acknowledgements

Sherman Chow is supported by the Research Grant Council, University Grants Committee, Hong Kong under the General Research Fund (CUHK 14210319).

The CUHK authors would also like to thank ASM Pacific Technology Ltd, Department of Information Engineering, and Faculty of Engineering at CUHK for their awards supporting related undergraduate summer/final-year projects. We also thank Chen Change Loy for his early suggestions.

This research is supported by the National Research Foundation, Singapore under its Strategic Capability Research Centres Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of National Research Foundation, Singapore. Part of the work was done while the last author was with CUHK.

This work has been presented as invited talks in the Seventh International Workshop on Security in Cloud Computing[4] (Chow 2019), held in conjunction with the Fourteenth ACM Asia Conference on Computer and Communications Security, and 2020 CUHK Conference on Financial Technology: "How FinTech Transforms Organizations – Theory & Practice."[5] We thank the respective organizers for their interests, and the audiences for their questions and comments.

Finally, our thanks go to the anonymous reviewers.

References

Arnautov, S.; Trach, B.; Gregor, F.; Knauth, T.; Martin, A.; Priebe, C.; Lind, J.; Muthukumaran, D.; O'Keeffe, D.; Stillwell, M.; Goltzsche, D.; Eyers, D. M.; Kapitza, R.; Pietzuch, P. R.; and Fetzer, C. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 689–703. USENIX Assoc.

Beaver, D. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*, 420–432. Springer.

Chandran, N.; Gupta, D.; Rastogi, A.; Sharma, R.; and Tripathi, S. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *EuroS&P*, 496–511. IEEE.

Chaudhari, H.; Rachuri, R.; and Suresh, A. 2020. Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning. In *NDSS*. Internet Society.

Chow, S. S. M. 2019. Can We Securely Outsource Big Data Analytics with Lightweight Cryptography? In *Security in Cloud Computing (SCC), co-located with AsiaCCS*, 1–1. Invited paper for keynote talk.

Cruz-Roa, A.; Basavanhally, A.; González, F. A.; Gilmore, H.; Feldman, M. D.; Ganesan, S.; Shih, N.; Tomaszewski, J.; and Madabhushi, A. 2014. Automatic detection of invasive ductal carcinoma in whole slide images with convolutional neural networks. In *Medical Imaging: Digital Pathology*, volume 9041, 904103. SPIE.

Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; and Narayanan, P. 2015. Deep Learning with Limited Numerical Precision. In *ICML*, 1737–1746. JMLR.org.

Hynes, N.; Cheng, R.; and Song, D. 2018. Efficient deep learning on multi-source private data. arXiv:1807.06689.

Janowczyk, A.; and Madabhushi, A. 2016. Deep learning for digital pathology image analysis: A comprehensive tutorial with selected use cases. *Journal of Pathology Informatics* 7.

Jia, Y. 2014. *Learning Semantic Image Representations at a Large Scale*. Ph.D. thesis, UC Berkeley, USA.

Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R. B.; Guadarrama, S.; and Darrell, T. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Multimedia (MM)*, 675–678. ACM.

Krizhevsky, A.; and Hinton, G. 2009. *Learning multiple layers of features from tiny images*. Tech Report.

Kunkel, R.; Quoc, D. L.; Gregor, F.; Arnautov, S.; Bhatotia, P.; and Fetzer, C. 2019. TensorSCONE: A Secure TensorFlow Framework using Intel SGX. CoRR abs/1902.04413.

Ma, J. P. K.; Tai, R. K. H.; Zhao, Y.; and Chow, S. S. M. 2021. Let's Stride Blindfolded in a Forest: Sublinear Multi-Client Decision Trees Evaluation. In *NDSS*. Internet Society.

Mohassel, P.; and Rindal, P. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*, 35–52. ACM.

Ng, L. K. L.; and Chow, S. S. M. 2021. GForce: GPU-Friendly Oblivious and Rapid Neural Network Inference. In *USENIX Security*. USENIX Association. To appear.

Orenbach, M.; Lifshits, P.; Minkin, M.; and Silberstein, M. 2017. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys*, 238–253. ACM.

Orenbach, M.; Michalevsky, Y.; Fetzer, C.; and Silberstein, M. 2019. CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves. In *USENIX ATC*, 555–570. USENIX Association.

Shaon, F.; Kantarcioglu, M.; Lin, Z.; and Khan, L. 2017. SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors. In *CCS*, 1211–1228. ACM.

Tai, R. K. H.; Ma, J. P. K.; Zhao, Y.; and Chow, S. S. M. 2017. Privacy-Preserving Decision Trees Evaluation via Linear Functions. In *ESORICS Part (II)*, 494–512. Springer.

Tramèr, F.; and Boneh, D. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *ICLR*.

Wagh, S.; Tople, S.; Benhamouda, F.; Kushilevitz, E.; Mittal, P.; and Rabin, T. 2021. FALCON: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *PoPETs* 2021(1): 188–208.

Wong, H. W. H.; Ma, J. P. K.; Wong, D. P. H.; Ng, L. K. L.; and Chow, S. S. M. 2020. Learning Model with Error - Exposing the Hidden Model of BAYHENN. In *IJCAI*, 3529–3535.

Yang, G.; Zhang, T.; Kirichenko, P.; Bai, J.; Wilson, A. G.; and Sa, C. D. 2019. SWALP : Stochastic Weight Averaging in Low Precision Training. In *ICML*, 7015–7024. PMLR.

[4] https://conference.cs.cityu.edu.hk/asiaccsscc/19

[5] http://www.cfe.cuhk.edu.hk/ftc2020