GOTEN: GPU-OUTSOURCING TRUSTED EXECUTION OF NEURAL NETWORK TRAINING AND PREDICTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Before we saw worldwide collaborative efforts in training machine-learning models or widespread deployments of prediction-as-a-service, we need to devise an efficient privacy-preserving mechanism which guarantees the privacy of all stakeholders (data contributors, model owner, and queriers). Slaom (ICLR '19) preserves privacy only for prediction by leveraging both trusted environment (*e.g.*, Intel SGX) and untrusted GPU. The challenges for enabling private training are explicitly left open – its pre-computation technique does not hide the model weights and fails to support dynamic quantization corresponding to the large changes in weight magnitudes during training. Moreover, it is not a truly outsourcing solution since (offline) pre-computation for a job takes as much time as computing the job locally by SGX, *i.e.*, it only works before all pre-computations are exhausted.

We propose Goten, a privacy-preserving framework supporting both training and prediction. We tackle all the above challenges by proposing a secure outsourcing protocol which 1) supports dynamic quantization, 2) hides the model weight from GPU, and 3) performs better than a pure-SGX solution even if we perform the pre-computation online. Our solution leverages a non-colluding assumption which is often employed by cryptographic solutions aiming for practical efficiency (IEEE SP '13, Usenix Security '17, PoPETs '19). We use three servers, which can be reduced to two if the pre-computation is done offline. Furthermore, we implement our tailor-made memory-aware measures for minimizing the overhead when the SGX memory limit is exceeded (cf., EuroSys '17, Usenix ATC '19). Compared to a pure-SGX solution, our experiments show that Goten can speed up linear-layer computations in VGG up to $40\times$, and overall speed up by $8.64\times$ on VGG11.

1 INTRODUCTION

While deep neural networks (DNN) can produce predictive models with unparalleled performance, its training phase requires enormous data as input. A single data owner may not possess enough data to train a good DNN. Multiple data owners, say, financial institutions, may want to collaborate in training DNNs. Yet, they are often expected to protect the privacy of the data *contributors*. This discourages any collaborative training over global-scale data which is otherwise promising (Cheng et al., 2019). Moreover, to perform prediction using a trained model, *queriers* need to submit their own *private* data (*e.g.*, medical history). Meanwhile, the model owners want to protect the confidentiality of the trained model in the prediction phase as well. The exposure of the (parameters of a) model (to queriers or a third-party cloud server) may reveal information about its training data (Fredrikson et al., 2015), deterring the participation of data contributors. Also, the model itself is of high commercial value. These concerns hinder the deployment of prediction as a service.

An increasingly popular approach to ensure privacy is using trusted execution environment (TEE) (Cheng et al., 2019; Tramèr & Boneh, 2019) and in particular trusted processors, *e.g.*, Intel Software Guard Extension (SGX). When a data provider sends some private data to a server equipped with SGX, it can initialize an *enclave* to receive the data in a confidential and authenticated way and subsequently operate on them. Even the untrusted server, who physically owns the enclave, cannot read or tamper the data inside the enclave. This paper investigates the following questions: *Can we support DNN training (and prediction) by using SGX and untrusted GPU while still preserving the privacy of all stakeholders? If so, how much speedup do we gain by using GPU?*

1.1 Our Baseline Approach: CaffeSCONE

Arnautov et al. (2016) propose SCONE, a secure container mechanism that allows developers to directly run applications in an SGX enclave with almost zero code change[1]. We combine SCONE with Caffe (Jia et al., 2014), an efficient open-source DNN framework, to build our baseline privacy-preserving DNN framework. CaffeSCONE satifies most of our privacy requirements, in particular, for the training data and client queries. While TensorSCONE (Kunkel et al., 2019) also employed SCONE (but with another DNN framework – TensorFlow (Abadi et al., 2016a), it is unfortunately not open source. The value of our CaffeSCONE implementation is to enable more benchmarking for insight in possible improvements (eventually achieved by our main result). Our results (referring to Section 4.2) show that CaffeSCONE's performance greatly suffer from the enclave's memory limit as it needs an inefficient mechanism to handle excessive use of memory not affordable by the enclave. Also, we found that using more threads and cores cannot improve the performance.

1.2 Our Proposed Framework: Goten

GPU-powered Secure-Computation By using SGX solely, CaffeSCONE is already orders of magnitude faster than the state-of-the-art cryptographic solutions (SecureML (Mohassel & Zhang, 2017), MiniONN (Liu et al., 2017), Gazelle (Juvekar et al., 2018), and DiNN (Bourse et al., 2018), while only SecureML supports training). Nevertheless, in general, CPU (with or without SGX) is not optimized for costly operations in DNN such as matrix multiplication. Using specialized hardware such as GPU for such computation is a common practice. However, SGX-enclaves cannot directly leverage GPU because its security guarantee is bounded within the CPU and fixed memory. It is unclear how CaffeSCONE (and other works including TensorSCONE, Chiron (Hunt et al., 2018), and MLCapsule (Hanzlik et al., 2018)) can leverage GPU without trusting it (or losing privacy).

The SGX+GPU mode of our framework, which we call Goten, enables an even more efficient approach. To the best of our knowledge, no existing work ever explored this possibility on *privacy-preserving training*. A recent work Slalom (Tramèr & Boneh, 2019) also uses GPU but it only offers prediction privacy. We follow the common practice in the cryptographic privacy-preserving training literature (SecureML, its subsequent work (Wagh et al., 2019), and other prior works (Nikolaenko et al., 2013a;b)) which employ non-colluding servers. Specifically, our framework uses three non-colluding GPU-enabled servers, two of them with a trusted processor. This setup appears to be necessary when the primary goal is to achieve privacy without heavyweight cryptographic tools. In practice, one can employ cloud service providers who are market competitors and value their reputations, or involve a government agency especially in healthcare/financial settings.

Taking Full Advantage of the Servers We choose to exploit the server-aided setting fully and employ one additional server when compared with SecureML. What this server does is to "bootstrap" the *secret sharing* and *triplets* (Beaver, 1991) across the two servers, which SecureML assumes such a bootstrap has been done in advance in an offline phase. Goten thus achieves a higher throughput without worrying that the offline preparation will be "exhausted" when the demand reaches its peak, which is also a hidden problem not addressed by Slalom. It also means Goten provides a "true" outsourcing solution – the time needed for securely outsourcing the job to the untrusted GPU is less than that for computing the job locally by the SGX *plus any time needed for pre-computation*. If desired, one may easily adapt our framework back to the two-server setting. (See Section 2.2.)

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 for the following reasons. First, the many matrix multiplications in neural network may scale up the output values quickly, easily exceeding the numeric limit of the data type. Second, there are functions that map values to a small interval, *e.g.*, $\mathrm{softmax}()$ and $\mathrm{sigmoid}()$, require high precision. To avoid these potential accuracy problems, we developed a *data-type conversion scheme*, again, for enjoying "the best of both worlds," *i.e.*, the benefit of accurate floating-point operations on trusted processors and efficient fixed-point operations on GPUs. Our experiment (Section 4) confirms that our framework preserves high accuracy.

[1]Without SCONE, developing a program using SGX enclaves impose laborious tasks to the developers: separately writing code for trusted library and untrusted application, defining interfaces between these two parts, and compiling them with Intel SGX SDK.

Memory-aware Implementation A naïve solution of overcoming the memory limit of SGX enclaves to rely on the Linux's paging provided by Intel SGX SDK. However, it imposes much performance overhead ranging from $10\times$ to $1000\times$ comparing to unprotected programs (Arnautov et al., 2016) for exiting the enclave mode and switching back after processing the untrusted memory. Hence, in our framework, we take extra measures to reduce the memory footprints by looking into our specific DNN operations and handle any needed memory swapping by the enclave itself.

1.3 Technical Contributions

Using both SGX and GPU for privacy-preserving training may sound straightforward, but we stress that we tackled a number of issues. To better understand the obstacles we solved, here we revisit how Slalom performs privacy-preserving *prediction* and why it fails to support training. The core idea of Slalom can be described in simple terms: first apply static quantization on an input x to be protected, then outsource the job of computing $f(x + r)$ to GPU by hiding x with a *blinding factor* r in $\mathbb{Z}_q$ (where q is a large prime). Since it focuses on linear layers, f is linear and hence $f(x + r) = f(x) + f(r)$. When SGX gets back $f(x + r)$, it performs "unblinding" using $f(r)$ and obtains $f(x)$. For such outsourcing to be possible, $f(r)$ should be precomputed. As simple as it may seem, Slalom needs to minimize the following three kinds of overheads – (i) the untrusted GPU needs to perform computation over $\mathbb{Z}_q$ for the security of the blinding trick, (ii) the communication between TEE and the untrusted GPU, and (iii) loading the precomputed unblinding factor $f(r)$ to TEE. Looking ahead, we will face even greater challenges regarding (i) and (ii). Slalom addresses (iii) by assumption – it was done in an offline stage before the TEE needs to process any query. If we just ask the SGX to compute it, computing $f(r)$ is of the same complexity as $f(x)$. Another way is to load them on-spot. It is again subjected to the memory limit and incurs the unwanted communication overhead. More importantly, it is insecure to ask the untrusted environment to compute $f(r)$.

There are five conceptual challenges remain unsolved by Slalom regarding training. 1) Dynamic quantization: Slalom explicitly left it as one of the open challenges. 2) DNN weights are fixed at inference time, but it is not for training. This further complicates the dynamic quantization issue since the weights fluctuate. 3) The pre-computation technique do not apply for training. In more details, the training function is actually parameterized by a publicly-known weight W, *i.e.*, $f_W(x)$ multiples x with W. Moreover, the weight changes after (a batch of) operations are processed for changing which makes $f_W(r)$ useless for another weight W'. 4) It is now apparent that Slalom does not protect the model weight W, which should be protected in private training (and "more private" prediction). This is also one of the open challenges left explicitly by Slalom. 5) The last one is a challenge unique to our solution in addressing the other challenges. In their usage, TEE and GPU are co-located. However, in our settings, we need to propose an outsourcing solution which is efficient enough even we are subjected to an even higher communication overhead between the servers.

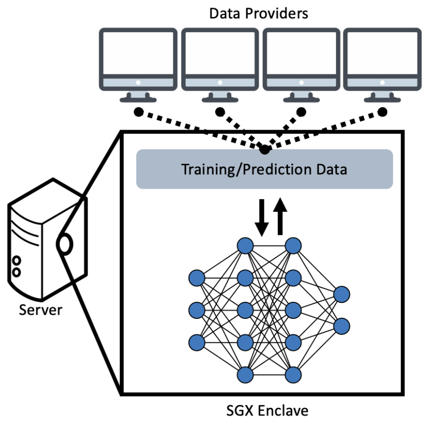
Goten is the first framework that preserve the privacy of not only the prediction queries but *the training data and model parameters* with GPU and trusted environment. Our work achieves the highest efficiency of training and prediction in such privacy setting. This is the also first work which performs extensive experimental investigations of this possibility. Concretely, in our case study on VGG, we can speed up a linear layers up to $40\times$, and improve the performance of VGG11 by $8.64\times$.

2 System Model

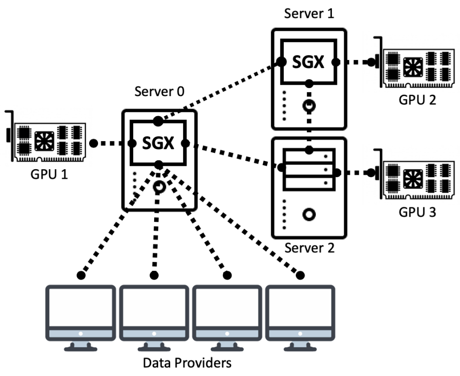
There are n mutually untrusted data providers who want to jointly train a DNN using their disjoint training data, but they are not willing to reveal their private data to others. They have already agreed on a specific DNN architecture. The corresponding code for the training algorithm is assumed to be genuine after manual or automated verification (Sinha et al., 2016). After training, queriers can use the resulting DNN to perform prediction such that the result is only revealed to the querier.

2.1 CaffeSCONE

The relationship between servers and the data providers are shown in Figure 1a. The server S initializes an enclave E with the specified program for training and prediction. The data providers $C_1, C_2, \cdots$ attest the enclave E, and verify that the enclave is running the intended program. Then they establish a secure channel with E in order to send their training data to the enclave E. The



(a) CaffeSCONE



(b) Goten

Figure 1: The Architecture of Goten and CaffeSCONE

enclave E then trains the neural network using the prescribed training algorithm. Once training is done, the data provider C_i sends his/her prediction query to the enclave E, which then computes the prediction result according to the trained model parameters θ.

2.2 Goten

This framework uses GPU to accelerate the computations of the fully-connected and convolutional layers. We introduce two additional non-colluding servers. Figure 1b illustrates the architecture.

The three servers S_0, S_1, and S_2 are equipped with GPU and SGX enabled processor. The server S_0 and S_1 initialize E_0 and E_1 respectively. All the enclaves would be attested by each others, the servers, and the data providers, and then build up secure channels. The server duties are not the same. S_0 and S_1 take care of DNN computations. S_2 merely provides multiplication triplets for linear computation, which are independent of the model parameters or the training/prediction data.

The training and prediction phase are similar to those in the pure SGX mode except two differences. To avoid cumbersome data transfer between the three servers, the data providers only need to send their data to E_0, which is then responsible for forwarding to other enclaves. We also significantly change the way of matrix multiplication to leverage the computational power of GPU. Instead of computing it in enclaves, we outsource the computation to GPU and protect the secret using additive secret sharing. We refer to Section 3.2 for the details of this protocol.

The attacker can compromise any subset of the data providers and at most one of the servers. Namely, two servers cannot collude with each other. We allow the attacker to control all the software (including operating system and hypervisor) of the server, but we assume it cannot launch any hardware attack on SGX. Denial-of-service or side-channel attacks are also out of the scope.

Our goal is to ensure that even such a powerful adversary cannot learn anything other than the DNN specification and the data of compromised parties. In particular, the parameters remains private. CaffeSCONE guarantees the correctness of both training and prediction. Goten does not provide it as we present it due to page limitation, but we can resort to the trick used by Slalom.

Reducing Non-colluding Servers Our design can be easily modified to use merely 2 servers with some preparation. The duty of S_2 is to produce two random matrices u, v, and the product $z = u \cdot v$, and distribute these matrices to E_0 and E_1. These enclaves can instead prepare u, v, and z by themselves, so S_2 is no longer needed. Similar tricks are also used by SecureML and MiniONN. Since matrix computation in enclaves is slower than that in GPU, E_0 and E_1 should pre-compute these matrices before the training/prediction process to prevent stalling the GPU Additional storage and preparation are required for removing S_2.

Moreover, the third server can also be a group of triplet providers which provide triplets in turns. In this case, these providers can amortize the computation requirement so they are not necessarily equipped with expensive GPUs and well-connected with the first two servers.

3 THE DESIGN OF GOTEN

3.1 HIGH-LEVEL IDEA

To improve the performance, we first outsource linear operations to GPU, and apply *SGX-aware chunked operations* to reduce the overhead caused by paging. An immediate difficulty of using GPU is how to preserve privacy because trusted execution environment on GPU does not exist. Our plan is to apply additive secret sharing to prevent leaking information to the hosts of GPU.

Still, CPU needs to convert data of linear layers into the format of additive secret sharing, and then convert the result from GPU back into the normal format for non-linear layers. We call these procedures pre-processing and post-processing of outsourcing linear operations. If they are not handled properly, the processing time could offset the performance gained from GPU. In the following sections, we will introduce our tricks for reducing the run-time of pre/post-processing, and present our modified secret sharing protocol that improves performance.

Moreover, not only the computation in layers but also pre/post-processing suffer from page faults. Hence, our SGX-aware chunked operations are vital for the performance. The high-level idea of chunked operations is to let the enclave specifies the piece of memory going to use, read and write the memory without triggering Linux's inefficient paging.

3.2 GPU-POWERED OPERATIONS VIA OUR OUTSOURCING PROTOCOL

GPU can speed up the computation of linear transformation and convolution by orders of magnitude. As explained in Appendix A.3, matrix multiplication and convolution occupy $\geq 90\%$ of computation time. Improving its efficiency is critical to the performance.

A trivial way is to encrypt a and b to the enclave and ask it to multiply them directly. Yet, it cannot leverage the batch-processing advantage of GPU and is inefficient for large scale computation. We aim to design a protocol which makes the best use of SGX and GPU without the shortcomings of either of them. Specifically, we leverage the SGX enclave to secure the unprotected computation environment of GPU, without the enclave performing any expensive decryption beyond the bare minimum, *i.e.*, two decryptions (for the two operands).

We start with the "bare minimum" operations which let the two enclaves E_0 and E_1 know the secrets a and b. The core design principle is to let the enclaves do what they are good for, *i.e.*, generating cryptographic randomness and using them to one-time pad some values. With the non-colluding assumption (required by the original protocol (Beaver, 1991)), we choose to fully exploit it and introduce one additional server to establish the triplets involved in computing $u \otimes v = z$. The triplets generation can be performed by "the initiating client" offline in existing protocols (Mohassel & Zhang, 2017; Liu et al., 2017), thus, this server can be removed as discussed in Section 2.2.

Figure 2 describes our protocol for outsourcing linear operation of $c = a \otimes b$. $\otimes$ can be convolution (so a and b are tensors) or matrix multiplication (for matrices a and b). Another important usage of enclaves is to store the same seed for deriving the random factors across all the servers. This trick forms a confidential channel between two servers very efficiently without AES or public-key encryption. For example, S_2 sends z in the form of $z - \text{Rand}(r_z)$ to E_0 and E_1 via insecure channels, which can be computed quickly. In other words, all instances of "$\rightarrow E_i : var$" in the figure refer to loading the variable(s) var to E_i directly without encryption.

The steps in line 3 of Figure 2 appear to be working on many more values than the trivial approach of computing $a \otimes b$. Our experiments in Section 4.2 confirms that the performance gain can be as large as $40\times$. Below, we discuss the changes we made over the original triplet-based protocol.

Parallelizable Pre-Processing without Communication Our protocol is still based on the existing secret-sharing based protocol (in Appendix A.4) but with improvement. Recall that the central idea is to compute $a \otimes b$ by operating over (e, f), which is a masked version (a, b). In the origi-

```
Secure Outsourcing of Linear Operation $\otimes$ to GPU
1 :   $S_2 : u \leftarrow \text{Rand}(r_u), v \leftarrow \text{Rand}(r_v), z = u \otimes v, \langle z \rangle_1 \leftarrow z - \text{Rand}(r_z)$
2 :   $S_2 \to E_0, E_1 : \langle z \rangle_1$
for $i = 0, 1$ in parallel:
3 :   $E_i : \langle a \rangle_i \leftarrow \text{Gen}_i(a, r_a), \langle b \rangle_i \leftarrow \text{Gen}_i(b, r_b), e = a - \text{Rand}(r_u), f = b - \text{Rand}(r_v),$
      $\langle z \rangle_0 \leftarrow \text{Rand}(r_z), K_{0\to1} \leftarrow \text{Rand}(r_{k_0}), K_{1\to0} \leftarrow \text{Rand}(r_{k_1})$ in parallel;
4 :   $E_i \to S_i : \langle a \rangle_i, \langle b \rangle_i, e, f, K_{i\to1-i}$
5 :   $S_i \to E_i : c_i = \langle a \rangle_i \otimes f + \langle b \rangle_i \otimes e - i \cdot e \otimes f$
6 :   $S_i \to E_{1-i} : C_{1-i} = c_i - K_{i\to1-i}$
endfor
7 :   $E_0 : c = c_0 + (C_0 + K_{1\to0}) + \langle z \rangle_0 + \langle z \rangle_1$
      $E_1 : c = c_1 + (C_1 + K_{0\to1}) + \langle z \rangle_0 + \langle z \rangle_1$
```

Figure 2: Protocol for Outsourcing Linear Operation $\otimes$

nal protocol, the shares $(\langle a \rangle_0, \langle b \rangle_0)$ and $(\langle a \rangle_1, \langle b \rangle_1)$ from the two parties (S_0 and S_1 here) must be masked independently by the corresponding one-time pads $(\langle u \rangle_0, \langle v \rangle_0)$ and $(\langle u \rangle_1, \langle v \rangle_1)$. After this step, they must interact to produce e and f.

In our protocol, both enclaves know a and b, so they can use the same seed to derive the same one-time pads u and v (which is in, say, $\mathbb{Z}_q^m$) and thus obtain e and f without any interaction. This saves half of the pre/post-processing and communication cost. This also make e and f no longer dependent on $\langle a \rangle_i$ and $\langle b \rangle_i$. Thus, all the steps in line 3 of Figure 2 can be done in parallel. The run-time of this pre-processing step is then further reduced roughly by $3/4$, *i.e.*, the cost is $1/4$ of the original. Furthermore, because E_0 and E_1 no longer need to interact until the very last step for result construction, they can also work in parallel.

Reducing Run-time of Share Reconstruction Unlike the original standalone protocol where each party only needs to learn a share $\langle c \rangle_i$ of c but not $c = a \otimes b$ itself, it is necessary for our enclaves to know c because they need to perform the succeeding non-linear operations of non-linear layers. (In some existing protocols , c is actually recovered "implicitly" via cryptographic means, say, within a garbled circuit.) A naïve way is to let S_i encrypt their respective shares to the other enclave E_{1-i}. Again, we use the common seed to form a secure channel which lets S_i one-time-pad its own share c_i into a ciphertext C_{1-i} for E_{1-i} via the key $K_{i\to1-i}$ derived from the seed. In total, we reduce pre/post-processing time by roughly 87.5% and halve the communication cost.

Performance Gain for Linear Layers Our outsourcing protocol, while optimized, still imposes overhead in pre/post-processing and communication between the servers. It is instructive to confirm how much we gain. Beyond the obvious reliance on the relative performance of the GPU, it turns out to be crucially relying on the shapes of the input and weight.

We first analyze the case of fully-connected layers. Assume $x \in \mathbb{Z}_q^{m\times k}$ is the input, $w \in \mathbb{Z}_q^{k\times n}$ is the weight, and $y \in \mathbb{Z}_q^{m\times n}$ is the output, We found that we should maximize $\min(m, k, n)$. Since m, the batch size, is usually small compared to k and n, it is better to be large.

Analysis. We should minimize the run-time ratio of our GPU-powered matrix multiplication scheme to the vanilla CPU scheme. The forward computation in fully-connected layer is $x \otimes w = y$. The run-time of our GPU-powered scheme is $t_{\text{pre-proc}} \cdot (m \cdot k + k \cdot n) + (t_{\text{post-proc}} + t_{\text{comm}}) \cdot (m \cdot n) + t_{\text{gpu-op}} \cdot (m \cdot k \cdot n)$.

The backward computation computes, $dx = dy \otimes w$ and $dw = dy^T \otimes x$, where dx, dw, and dy are the gradient of x, w, and y respectively, and they are of the same size of their counterpart. Similar to the run-time analysis above, we can derive that the total run-time of both the forward and backward computation is $t_{\text{gpu-scheme}} = (2 \cdot t_{\text{pre-proc}} + t_{\text{post-proc}} + t_{\text{comm}}) \cdot (m \cdot k + k \cdot n + m \cdot n) + 3 \cdot t_{\text{gpu-op}} \cdot (m \cdot k \cdot n)$. For our ease, we denote $t_{\text{extra}} = 2 \cdot t_{\text{pre-proc}} + t_{\text{post-proc}} + t_{\text{comm}}$. Also, the run-time of the vanilla CPU scheme is $t_{\text{cpu-scheme}} = 3 \cdot t_{\text{cpu-op}} \cdot (m \cdot k \cdot n)$.

Finally, the run-time ratio of these two schemes is

$$\frac{t_{\text{gpu-scheme}}}{t_{\text{cpu-scheme}}} = \frac{t_{\text{extra}}}{t_{\text{cpu-op}}} \cdot (\frac{1}{m} + \frac{1}{n} + \frac{1}{k}) + \frac{t_{\text{gpu-op}}}{t_{\text{cpu-op}}}.$$

The last term matches with the intuition that GPU governs the performance gain. The pre/post-processing and communication time also play an important role if $1/m + 1/n + 1/k$ is large. Note that the inverse of $1/m + 1/n + 1/k$ is also known as the *arithmetic intensity* (cud, 2019).

The analysis on convolution layers follows the same principle but is more involved. If we assume the image size of input and output are the same, we can have a similar result as fully-connected layers by replacing m, k, and n to $C_{out} \cdot f_h \cdot f_w$, $C_{in} \cdot f_w \cdot f_h$, and $B \cdot I_h \cdot I_w$. Figure 5a shows convolution gains speed-up as expected when paging overhead is low.

3.3 Data Types and Dynamic Quantization

The triplet trick we used operates over fixed-point numbers in $\mathbb{Z}_q$, while common neural network framework operates over float-point numbers. Therefore, Goten has to accommodate the fixed-point setting so that it can attain superior performance as if using float-point numbers.

The choice of $\mathbb{Z}_q$ GPU is slow in modular arithmetic, off-the-shield optimized libraries do not support them. To work on $\mathbb{Z}_q$ integers, we thus put them as floats as Slalom (Tramèr & Boneh, 2019). This leaves us only 53 significant bits plus a sign bit to represent the integers in linear layers (where the rest of $(64 - 53 - 1)$ exponent bits are 0).

To make sure the result of 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 columns of matrix a or the $C_{in} \cdot f_w \cdot fw$ in convolution.

To avoid overflow in $\mathbb{Z}_q$, q should be large; but predicting the value of k beforehand is hard. We thus resort to the heuristics of testing different choices of q over common VGG networks. Based on our experiments, $q = 2^{21} - 9$ is the largest value that does not overflow in almost all ($\approx 100\%$) cases.

Challenges in Quantization To compute $x \otimes_{\text{f}} w$ with floating-point multiplication $\otimes_{\text{f}}$, we need a quantization scheme to convert floats to fixed-point numbers and vice versa for linear layers. We first quantize x and w into $x_Q = Q(x; \theta_x)$ and $w_Q = Q(w; \theta_w)$, where θ_x and θ_w are the corresponding quantization parameters. 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 $y = Q^{-1}(y_Q; \theta_x, \theta_w) \approx x \otimes_{\text{f}} w$.

Slalom only supports prediction. Knowing the model, it knows the value distribution of model parameters. It can then derive the distribution of the input, output, and intermediate values. Picking a *static* scaling parameter that minimizes the error in prediction is thus relatively easy. In Slalom, $Q(\cdot; \theta)$ is always parameterized by $\theta = 2^8$ for all data (inputs and weights) and every computation. In short, static quantization may not pose a big problem in a prediction-only framework.

Dynamic Quantization for Training Slalom clearly states that quantization for training is a challenging problem. For training, the range of gradient of the weight may change, hence the output, and the input of the successive layer. Knowing the value distribution prior to training is hard, so we cannot determine what parameters for Q is good enough to support training.

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 has to be *efficient* since it is part of the pre(/post)-processing of our GPU-powered scheme. An inefficient scheme would reduce or even offset the performance gain.

Our Choice SWALP (Yang et al., 2019) is a training scheme which works in a low-precision setting. The forward and backward computation are performed in low-precision fixed-point, but the weights are stored and updated in floats with high-precision.

Suppose bit is the number of bits available for the low-precision computation, and the default value is 8. For both the weight and the input, SWALP first finds out the maximum absolute value, and then calculates its exponent in the format of bits, *i.e.*, compute $\mathsf{exp} = \lfloor (\log_2 \circ \max \circ \text{abs})(data) \rfloor$.

Then, it scales up all the value by that exponent so that the new maximum values are roughly aligned to $2^{\mathsf{bit}} - 2$, rounds them up stochastically (Gupta et al., 2015), and clips all the value to $[-2^{\mathsf{bit}} - 1, 2^{\mathsf{bit}} - 1 - 1]$, *i.e.*, $data_Q = Q(data, \mathsf{exp}) = \mathsf{clip}(\lfloor data \cdot 2^{-\mathsf{exp}+\mathsf{bit}-2} \rceil)$. After the computation, the resulting values are scaled down accordingly, *i.e.*, $y = y_Q \cdot 2^{\mathsf{exp}_x + \mathsf{exp}_w - 2 \cdot \mathsf{bit} + 2}$

Based on the existing SWALP experiment, its accuracy drops by less than 1% when compared to training in full-precision for VGG16, and the operands are only of 8 bits. Also, finding the maximum absolute value and scaling up and down the values only requires 3 linear scans. The scaling can be fused with other pre/post-processing too. Finally, this scheme matches with our expectation that it is dynamic because it samples the maximum value of the weight and input every iteration. Section 4.2 shows that with this quantization scheme, Goten can train VGG11 to attain high accuracy efficiently.

3.4 Memory-aware Measures

When the allocated memory in enclave access the limit of 128MB, it incurs excessive overhead. We apply a memory-aware mechanism for handling most operations in enclave to mediate this problem.

A naïve solution is Linux's paging, which is provided by Intel SGX SDK. However, native paging is known to be inefficient. As reported in SCONE (Arnautov et al., 2016), memory access can be $10 - 1000\times$ slower compared to plaintext setting. Eleos (Orenbach et al., 2017) explains that triggering SGX native paging would make the CPU core exit the enclave mode, which is time-consuming. The more memory allocated, the more frequent such expensive operations are invoked.

To prevent this expensive operations, our SGX-aware chunked operation restricts the enclave's memory space to 128MB so that it would not trigger the nav̈e paging. When Goten needs to allocate memory more than 128MB, it would directly handling encrypt the chunk of memory and evict it in untrusted zone, which, unlike the naïve paging, does not leave the enclave mode. When it need to use memory not in the enclave, it loads the chunk of memory into the enclave and decrypt it. Section 4.2 shows that our mechanism speed up the computation of non-linear layers by $8.72\times$.

When implementing operations inside the enclave, we are aware of this mechanism and try to minimize the memory access across the boarder between the trusted/untrusted zone. Some ways are fusing operations that use the same set of memory and independently handling batches in non-linear layers to prevent excessive use of memory.

Eleos (Orenbach et al., 2017) is also another mechanism for mediating page-fault overhead. It allows the program to handle page-fault without exiting the enclave, CoSMIX (Orenbach et al., 2019) further automate the instrument for this paging-handling mechanism. However, its implementation was release less than a month so we have not compared or integrated with it.

4 Empirical Evaluation

For Goten, its SGX part is written in C++ and compiled with Intel SGX SDK 2.5.101.50123, and we use Pytorch (pyt, 2019) 1.2 on Python 3.6.9 to marshal network communication and operation on GPU, which run with CUDA 9.0. The C++ code is compiled by GCC 7.4. Also, we reuse some code of Slalom (Tramèr & Boneh, 2019), including their code of crypgtographicially-secure random number generation and encryption/decryption, and their OS-call-free version of Eigen, a linear-algebra library. All the experiments were conducted for at least 5 times, and we report the average of the results. We uploaded our source code to `https://redacted-for-submission`.

4.1 Setup

SGX's Simulation Mode and Hardware Mode Only limited models of Intel CPU are powered by SGX, which can run in the regular *hardware mode* and enjoy the SGX protection. Intel SGX SDK also provides *simulation mode* for testing purpose. Its code compilation is almost the same as hardware mode except that the program is not protected by SGX, which is fine for our purpose since the DNN training and prediction algorithms are *publicly known*. In term of performance, the largest difference between these two mode is related to paging. When the allocated memory in enclaves exceeds its physical limit, the enclaves in hardware mode may suffer much larger overhead compare to native programs. In simulation mode, the overhead is little.

Experiemental Environment for CaffeSCONE and Goten We evaluate the performance CaffeSCONE on a computer (which supports SGX hardware mode) equipped with Intel i7-7700 Kaby Lake Quad-cores 4.3GHz CPU and 16GB RAM, using Ubuntu 18.04. For reproducibility and for the ease of setting up the experiment, we evaluate the performance Goten on 3 Google Cloud VMs. We specify all VMs to equip CPU with Sky Lake, the latest microarchitecture can be used for Google Cloud's VM. Unfortunately, all CPUs on Google VMs do not support Intel SGX's hardware mode. Also, all these machines are equipped with 32GB RAM and a Nvidia V100 GPU.

Calibration on experiment results The experiments on the environment of Goten would underestimate the performance of programs running in SGX simulation mode because the CPUs have lower clock rate and older microarchitecture compared to Intel i7-7700.

To make the comparison between these two frameworks fair, we calibrate Goten's CPU runtime to CaffeSCONE's CPU runtime. We measure the runtime of the non-linear layers in the two above-mentioned environments. We found that the experiment Goten's environment would overestimate the runtime on CaffeSCONE's CPU. Hence, we decide to scale down the runtime of most time-consuming non-linear layers in Goten according to the data collected. The scaling factor for ReLU is 0.96, for Batchnorm is 0.56, for Maxpool is 0.85.

Since the runtime in linear layers related to the transfer between CPU and GPU and over the network, it is hard to calibrate the runtime between on CPU data solely. Also, our data show that the pre/post-processing CPU time is similar on hardware mode and simulation mode. Hence, we do not calibrate the runtime of linear layers. Tables 1 and 2 and Fig. 4 are calibrated by this method.

Choice of Dataset and Architecture: CIFAR-10 and VGG11 Both of Goten and CaffeSCONE are evaluated on CIFAR-10, a common dataset for benchmarking the accuracy. We pick a VGG architecture with 11 layers and batch normalization layers for our experiments because it is typical DNN which can attain high accuracy on CIFAR-10. Also, VGG11 is small so CaffeSCONE, a CPU-only framework, can easily handle it.

4.2 PERFORMANCE ON VGG11

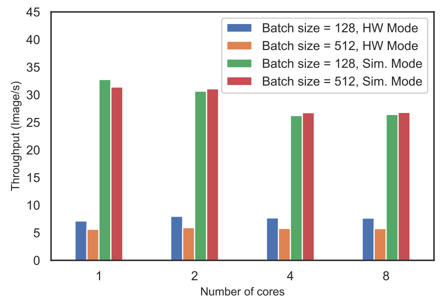


Figure 3: Training Throughput of CaffeSCONE

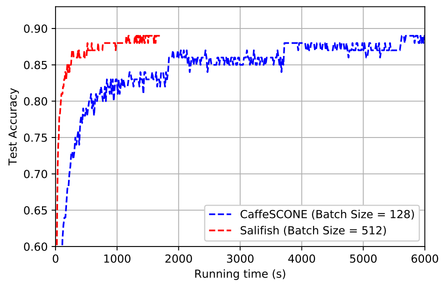


Figure 4: Accuracy convergence in VGG11

Throughput of CaffeSCONE Fig. 3 show the performance of CaffeSCONE with different batch size, execution mode, number of cores. It shows that hardware mode is about 4× slower than simulation mode. Also, using more cores does only have slight impact on the performance, or even harm the performance. To be fair, we pick the peak performance of CaffeSCONE, which is 32 image/s in simulation mode and 8 image/s in hardware mode. Both of the batch sizes are 128.

Training Throughput of Goten The latency of Goten running a forward-backward iteration is 7.4s when batch size is 512. Its throughput is 69 image/s. We achieve 8.6× improvement. Fig. 1 shows the time distribution of linear and non-linear layers. Goten greatly speeds up the linear layers in VGG11 by 8.59× and non-linear layers by 8.72×, and in total 8.64×

Convergence on Quantized Neural Netowrks Fig. 4 shows that the convergence trajectory of Goten and CaffeSCONE. For fair comparison, we set the batch size of CaffeSCONE as 128 as a

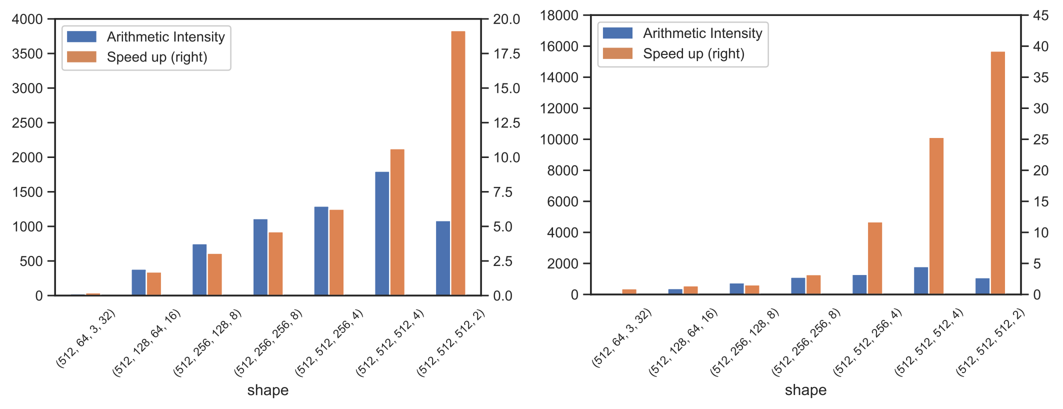
Table 1: Time Distribution on Linear/Non-linear Layers

	Linear Layers		Non-linear Layers		Total
	Time (ms)	Proportion	Time (ms)	Proportion	Time (ms)
CaffeSCONE (BS=128)	9243	57.7%	6774	42.3%	16017
Goten (BS=512)	4306	58.1%	3106	41.9%	7412
Speed-up	8.59×	-	8.72×	-	8.64×

Table 2: Attaining accuracy using GPU-powered Scheme

Accuracy	0.90	0.89	0.88	0.87	0.86	0.85
Speed up	-	4.93	7.28	7.31	7.31	11.78

typical floats setting which achieves the highest throughput. We run Goten with batch size of 512 to gain performance. Goten can attain high accuracy on our test data faster than CaffeSCONE. Table 2 shows the speed up which ranges from $4.93\times$ to $11.78\times$. It shows our quantization scheme does not have significant impact on training, and it attain a high accuracy in a shorter time. However, Goten still cannot attain 0.9 accuracy after 200 epochs, while CaffeSCONE can.



(a) With Low Paging Overhead (b) SGX Hardware Mode

Figure 5: Speed up vs. Arith. Intensity of GPU-powered Conv. of Shape $(B, C_{out}, C_{in}, I_{hw})$

Micro-benchmarks: Speedup of Our GPU Outsourcing Protocol Fig. 5 show that the speed-up and arithmetic intensity of each convolution layers presented in VGG with CIFAR-10. The shapes correspond to (the batch size, the number of input channels, the number of output channels, the height and width of input images). The filter size of all layers are 3×3.

Fig. 5a shows the the result in simulation, where paging overhead is low. The result confirms with our analysis in Section 3.2: the higher arithmetic intensity a convolution layer has, the more performance it gains. Furthermore, to have performance gain in our experimental environment, the arithmetic intensity should be at least 250. Also, we notice that the layer with image size 2×2 actually has huge performance gain while it has relatively low arithmetic intensity. We suspect that it is because Caffe cannot efficient handling input with small image size in CPU.

Fig. 5b shows the speedup in hardware mode, where paging overhead is huge. It shows much high speed up when there are small the images and many the input channels, and the speed up is not proportional to the architecture intensity. We suspect that the convolution's implementation of Caffe amplify the paging overhead in the above-mentioned situation.

Fig. 1 shows that Goten totally speed up the linear layers in VGG11 by $8.6\times$,

5 CONCLUSIONS

We proposed a new secure neural network framework using trusted processors. Our framework not only outperforms cryptographic solutions by orders of magnitude, but also resolved the memory limits issues in the existing state-of-the-art trusted processors approach (Ohrimenko et al., 2016). We made privacy-preserving training, prediction, and model-outsourcing for very deep neural networks more deployable in practice by advancing the frontier of the SGX-based machine-learning. For the first time, we can run a very deep neural network, with privacy, but without any memory issue.

REFERENCES

Deep Learning Performance Guide :: Deep Learning SDK Documentation. `https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html`, 2019. [Online; accessed 13-Sept-2019].

PyTorch. `https://pytorch.org`, 2019. [Online; accessed 13-Sept-2008].

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pp. 265–283, 2016a.

Martín Abadi, Andy Chu, Ian J. Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *CCS*, pp. 308–318. ACM, 2016b.

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. Performance, design, and autotuning of batched GEMM for GPUs. In *ISC*, pp. 21–38, 2016.

Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: secure linux containers with intel SGX. In *OSDI*, pp. 689–703, 2016.

Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. Secure multiparty computation from SGX. In *Financial Crypt.*, pp. 477–497, 2017.

Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, pp. 420–432, 1991.

Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *CRYPTO*, 2018.

Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *USENIX Workshop on Offensive Technologies*, 2017.

Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R. Pietzuch, and Rüdiger Kapitza. SecureKeeper: Confidential zookeeper using Intel SGX. In *Middleware*, pp. 14, 2016.

Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. In *CCS*, pp. 486–497, 2007.

Somnath Chakrabarti. SGX memory oversubscription, 2017. `http://caslab.csl.yale.edu/workshops/hasp2017/HASP17-05_slides.pdf`.

Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *IEEE EuroS&P*, pp. 185–200, 2019.

Sherman S. M. Chow, Jie-Han Lee, and Lakshminarayanan Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *NDSS*. ISOC, 2009.

Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint 2016/086, 2016.

Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*. ISOC, 2015.

Cynthia Dwork. Differential privacy. In *ICALP*, pp. 1–12. Springer, 2006.

Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT*, pp. 486–503. Springer, 2006a.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pp. 265–284. Springer, 2006b.

C. Feng. SGX protected memory limit in SGX, 2017. `https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/670322`.

Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS*, pp. 1322–1333. ACM, 2015.

Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, pp. 201–210, 2016.

Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN 978-0-262-03561-3.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, pp. 1737–1746, 2015.

Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. MLCapsule: Guarded offline deployment of machine learning as a service. CoRR abs/1808.00590, 2018.

Danny Harnik and Eliad Tsfadia. Impressions of Intel SGX performance, 2017. `https://link.medium.com/ZiaueRi94Z`.

Susan Hohenberger and Anna Lysyanskaya. How to securely outsource cryptographic computations. In *TCC*, pp. 264–282. Springer, 2005.

Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. CoRR abs/1803.05961, 2018.

Intel. Intel Software Guard Extensions (Intel SGX), 2017. `https://software.intel.com/en-us/sgx`.

Geetha Jagannathan and Rebecca N. Wright. Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In *SIGKDD*, pp. 593–599, 2005.

Yangqing Jia. *Learning Semantic Image Representations at a Large Scale*. PhD thesis, University of California, Berkeley, USA, 2014.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM International Conference on Multimedia, MM*, 2014.

Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *USENIX Security*, pp. 1651–1669, 2018.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.

Roland Kunkel, Do Le Quoc, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. Tensorscone: A secure tensorflow framework using intel SGX. CoRR abs/1902.04413, 2019.

Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In *CCS*, pp. 619–631. ACM, 2017.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pp. 190–200. ACM, 2005.

Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *IEEE S&P*, pp. 19–38. IEEE, 2017.

Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *CCS*, pp. 801–812. ACM, 2013a.

Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P*, pp. 334–348. IEEE, 2013b.

Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, pp. 619–636. Usenix, 2016.

Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *EuroSys*, pp. 238–253. ACM, 2017.

Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves. In *USENIX ATC*, pp. 555–570, 2019.

Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Trans. Information Forensics and Security*, 13(5):1333–1345, 2018.

Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. SGX-BigMatrix: A practical encrypted data analytic framework with trusted processors. In *CCS*, pp. 1211–1228. ACM, 2017.

Reza Shokri and Vitaly Shmatikov. Privacy-preserving deep learning. In *CCS*, pp. 1310–1321. ACM, 2015.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *PLDI*, pp. 665–681. ACM, 2016.

Aleksandra B. Slavkovic, Yuval Nardi, and Matthew M. Tibbits. Secure logistic regression of horizontally and vertically partitioned distributed databases. In *ICDM*, pp. 723–728, 2007.

Raymond K. H. Tai, Jack P. K. Ma, Yongjun Zhao, and Sherman S. M. Chow. Privacy-preserving decision trees evaluation via linear functions. In *ESORICS*, pp. 494–512, 2017.

Qiang Tang and Husen Wang. Privacy-preserving hybrid recommender system. In *AsiaCCS-SCC*, pp. 59–66, 2017.

Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. CoRR abs/1810.00602, 2018.

Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *ICLR*, 2019.

Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *USENIX Security*, pp. 601–618, 2016.

Jaideep Vaidya, Hwanjo Yu, and Xiaoqian Jiang. Privacy-preserving SVM classification. *Knowl. Inf. Syst.*, 14(2):161–178, 2008.

Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on GPUs. In *OSDI*, pp. 681–696, 2018.

Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, 2019.

Boyang Wang, Ming Li, Sherman S. M. Chow, and Hui Li. A tale of two clouds: Computing on data encrypted under multiple keys. In *IEEE CNS*, pp. 337–345. IEEE, 2014.

Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for Intel SGX enclaves. In *Middleware*, 2018.

David J. Wu, Tony Feng, Michael Naehrig, and Kristin E. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016(4):335–355, 2016.

Guandao Yang, Tianyi Zhang, Polina Kirichenko, Junwen Bai, Andrew Gordon Wilson, and Christopher De Sa. Swalp: Stochastic weight averaging in low-precision training. arXiv:1904.11943, 2019.

Hwanjo Yu, Jaideep Vaidya, and Xiaoqian Jiang. Privacy-preserving SVM classification on vertically partitioned data. In *PAKDD*, pp. 647–656, 2006.

Yongjun Zhao and Sherman S. M. Chow. Privacy preserving collaborative filtering from asymmetric randomized encoding. In *Financial Crypt.*, pp. 459–477, 2015.

A Preliminaries

A.1 Neural Networks

A neural network gains its predictive power by imitating biological neural networks (Goodfellow et al., 2016). A (feedforward) neural network can be represented by a sequence of transformations.

This paper focuses on supervised learning — every training data is a data point $\mathbf{x}$ associated with a label $\mathbf{y}$, and the neural networks try to learn the relationship between $\mathbf{x}$ and $\mathbf{y}$. Prediction in supervised learning outputs a label of query $\mathbf{x}$.

We refer the computation for prediction by *forward-propagation*. For training, gradient descend is usually employed, where the computation for updating the parameters is called *backward-propagation*.

A.1.1 Common Layers in Neural Networks

Roughly, transformations in a neural network can be divided into two categories: linear transformation and non-linear transformation.[2]

For the linear transformation, we have two kinds of layers.
i) *Fully-connected layer* (a.k.a. *dense layer*) — It just multiplies a weighting matrix to the input (for training or prediction).

ii) *Convolutional layer* — It is similar to the convolution operation except it rotates the kernels by 180 degrees. The data-structure of inputs, outputs, and kernels are *tensors*, which are usually $3/4$-dimensional.

For non-linear transformation, we have —
i) *Activation layer*, which applies a non-linear function on each element to mimic the impulse activation of biological cells.
ii) *Pooling layer*, which aggregates values in a group after applying a function like $\max()$ or $\mathrm{mean}()$

[2]Some weird layers may appear in some architecture but can be easily implemented using the principle we introduced.

function.

iii) *Output layer*, which outputs the results in the prediction phase. In the training phase, it computes a loss value measuring the error between the ground truth and the neural network's prediction.

A.1.2 Computational Aspects

The linear transformation is the most computationally intensive part (Jia, 2014) when we compute in plaintext. The same applies to the SGX setting. Looking ahead, we will leverage GPU to accelerate the computation of linear layers. Looking ahead, we further outsource the linear transformation to multiple servers by additive secret sharing (Section A.4) to improve efficiency.

In SGX-enclave, the non-linear transformation can be processed in plaintext efficiently. These non-linear transformations are basically aggregating the output from its previous layer and/or applying element-wise operations. A simple but efficient way to handle them is to load the entries from the previous layer to the enclave cache memory one-by-one in a deterministic order and output the results once it got enough inputs. In this way, the data remains confidential and the memory access pattern is hidden.

In contrast, without SGX, (cryptographic) solutions either use garbled circuits, resulting in high computation and communication overhead (SecureML (Mohassel & Zhang, 2017), MiniONN (Liu et al., 2017), and Gazelle (Juvekar et al., 2018)), restricted choice of the activation layer and pooling layer (CryptoNet (Gilad-Bachrach et al., 2016)), or dramatic reduction of the size of neural networks (DiNN (Bourse et al., 2018)). As a result, these solutions are not compatible with many well-developed neural network architectures such as AlexNet (Krizhevsky et al., 2017), VGG16/19 (Simonyan & Zisserman, 2015), *etc.*

A.1.3 Very Deep Convolutional Network (VGG)

This is a family of very deep neural networks with $9 - 19$ layers with parameters (Simonyan & Zisserman, 2015) and has extraordinary performances on object classification. They have convolution layers with similar setting, *e.g.*, all of the convolution has filters of 3×3 and followed by ReLU and some of them further followed by 2×2 max-pooling layers. They are commonly used neural networks and hence it is worth to study how to improve the performance of neural networks in privacy-preserving setting.

A.2 Intel SGX

SGX is the latest Intel hardware-assisted remote secure computing design. Since its seventh generation (Intel, 2017), Intel introduced a set of instructions and hardware design with which an *enclave* can be allocated in the trusted hardware, protecting the privacy and integrity of the data to be processed within it.

A.2.1 Security Enclaves and Memory Limit

In SGX, enclaves are used as secure containers. When the secure software requests a secure container, an enclave will be loaded with the code and the data specified by the secure software. The enclave will isolate itself from the rest of the computer. Then the data owner can verify the integrity of the enclave by undergoing a standard remote attestation of SGX. Inside an enclave, all the data will be stored in the main memory in an encrypted and authenticated form when the CPU core is not processing them. When some specific data is going to be processed, it will be loaded into memory caches dedicated to a CPU core with SGX protection enabled and then be decrypted.

Although Intel claims that the current SGX supports up to 128MB of memory, at most 90MB is usable according to Shaon et al. (2017).

A.2.2 Generic Application

The trusted hardware is directly applicable to secure computation. Imagine that a data provider holding some sensitive data wants to perform some secure computation on a remote server. The data provider does not trust the server owner and thus he wants that only the server owner can know the pre-defined output. The trusted processor is an efficient solution satisfying these requirements: data can be processed in plaintext inside the trusted processor but remains unknown and tamper-proof, even to the server owner. Of course, the data owner needs to trust both the software provider and the hardware manufacturer.

A.3 Graphics Processing Unit

A GPU consists of thousands of cores that can perform similar instructions in parallel. If an algorithm is parallelizable, GPU can increase its computation performance by orders of magnitude.

The most computationally intensive part of neural networks can be transformed into matrix computation, which is well-suited for GPU. Jia (2014) showed that fully-connected layers and convolutional layers occupy over 95% computational time. Abdelfattah et al. (2016) concluded that GPU can speed-up matrix multiplication by $\geq 10\times$ compared to multi-core CPU.

A.4 Two-Party Computation via Secret Sharing

For two servers P_0 and P_1 holding private input $a, b \in \mathbb{Z}_q$ respectively, where q is a prime, they can let a third server learn $c = a + b \in \mathbb{Z}_q$ without revealing a, b as follows. P_0 chooses a uniformly random $a' \in \mathbb{Z}_q$, then sends $\langle a \rangle_1 = a'$ to P_1, and keeps $\langle a \rangle_0 = a - a'$. P_1 does a similar job: samples and sends $\langle b \rangle_1 = b'$ to P_0, and keeps $\langle b \rangle_0 = b - b'$. No one revealed a or b in this process. Then, P_0 computes $\langle c \rangle_0 = \langle a \rangle_0 + \langle b \rangle_0$ and P_1 computes $\langle c \rangle_1 = \langle a \rangle_1 + \langle b \rangle_1$. At this point, P_0 and P_1 both hold (additive) secret shares of $c = a + b$. Any third party with both shares $\{\langle c \rangle_i\}$ can learn $c = \langle c \rangle_0 + \langle c \rangle_1$.

Beaver (1991) generalized the above method to let P_0 and P_1 compute secret shares of $c = a \cdot b$ as follows. Suppose P_0 and P_1 have already pre-computed additive secret shares of u, v, and z where $u \cdot v = z$. Namely, P_i has $\langle u \rangle_i$, $\langle v \rangle_i$, and $\langle z \rangle_i$. P_i masks $\langle a \rangle_i, \langle b \rangle_i$ via $\langle e \rangle_i = \langle a \rangle_i - \langle u \rangle_i$ and $\langle f \rangle_i = \langle b \rangle_i - \langle v \rangle_i$. They then exchange $\langle e \rangle_i$ and $\langle f \rangle_i$ to reconstruct e and f, which is masking a and b respectively. 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 $\langle c \rangle_0 + \langle c \rangle_1 = ab$. This technique can be further generalized to matrix addition/multiplication by replacing $\mathbb{Z}_q$ with $\mathbb{Z}_q^{m \times k}$ or $\mathbb{Z}_q^{k \times n}$. Indeed, this technique can applied to any linear operation, including convolution.

Using this protocol as-is requires two rounds of communication (for recovering (e, f)) and pre-computation (of shares of (u, v, z)). Looking ahead, we will illustrate how to reduce the communication cost and the pre-computation and hence improve the throughput.

In the rest of the paper, we use $\text{Rand}(r_x)$ to denote a function that takes a random seed r_x and outputs a random element $x' \in \mathbb{Z}_q$. Then the (additive) secret share of x held by P_i can be written as $\langle x \rangle_i = \text{Gen}_i(x, r_x) = i \cdot x + (-1)^i \cdot \text{Rand}(r_x)$.

B Related Work

B.1 Cryptographic Solutions

Gilad-Bachrach et al. (2016) proposed CryptoNet. It exploits non-linear functions supported by levelled homomorphic encryption (LHE) and parallel computation to improve the efficiency of neural network evaluation. However, it only supports limited activation function (x^2 or $\text{sigmoid}(x)$) and pooling function (average pooling). The experiment results of CryptoNet showed that it is roughly $1000\times$ slower than running a similar neural network in plaintext,

Subsequent works (Mohassel & Zhang, 2017; Liu et al., 2017; Juvekar et al., 2018) improve or extend CryptoNet in various dimensions. SecureML (Mohassel & Zhang, 2017) uses two non-colluding servers to support both training and prediction for neural networks, but it is slower than CryptoNet in prediction. MiniONN (Liu et al., 2017) achieves higher prediction accuracy than SecureML for the same network structure. It is also $5\times$ faster than SecureML for small neural networks with single instruction multiple data (SIMD) batching technique on HE.

To the best of our knowledge, Gazelle (Juvekar et al., 2018) is the state-of-the-art cryptographic approach in terms of latency. It performs much better than CryptoNet/MiniONN by delicately choosing the HE scheme with optimized parameters to fit the hardware architecture. Gazelle has much lower latency than MiniONN/SecureML as its plaintext space is at most 20 bits. However, it is still unclear whether Gazelle harms the accuracy which is not stated in their paper (Juvekar et al., 2018).

DiNN (Bourse et al., 2018) follows an approach similar to CryptoNet's. It does not require user interaction during the evaluation. To the best of authors' knowledge, it is the state-of-the-art pure-HE-based approach. Yet, as stressed in the DiNN paper (Bourse et al., 2018), their aim is to show that a pure-HE approach is possible and can outperform CryptoNet, at the cost of lower accuracy.

In general, all frameworks mentioned above use expensive cryptographic primitives, such as LHE, GC, and OT, during (training and) prediction, resulting in huge data and computation overheads. Also, using these primitives usually requires multiple rounds of communication between different parties.

As a final remark, there are cryptographic solutions that protect the privacy of (mostly the "prediction" phase of) other machine learning algorithms. A non-exhaustive list includes decision trees (Tai et al., 2017; Wu et al., 2016; Bost et al., 2015), logistic regression (Slavkovic et al., 2007; Bost et al., 2015), support vector machine (Vaidya et al., 2008; Yu et al., 2006), collaborative filtering (Tang & Wang, 2017; Zhao & Chow, 2015), and k-means clustering (Bunn & Ostrovsky, 2007; Jagannathan & Wright, 2005). They are conceivably less powerful than a deep neural network.

B.2 Trusted Execution Environment

Ohrimenko et al. (2016) proposed data-oblivious machine learning algorithms using SGX for training and prediction. Their work also defends against some potential side-channel attacks using oblivious operations. However, their algorithms cannot handle any layer of size exceeding the amount of usable memory (90MB) in an enclave.

The memory limit has been a huge drawback of SGX. Different efforts have been devoted to resolving this issue. Shaon et al. (2017) proposed SGX-BigMatrix. It supports operations on matrices which size exceed 90MB, but still have very high overhead comparing to optimized libraries for unprotected matrices. Linux's SGX supports memory oversubscription for enclaves, but it introduces overhead for paging, which is reported widely (Weichbrodt et al., 2018; Chakrabarti, 2017; Harnik & Tsfadia, 2017; Brenner et al., 2016; Arnautov et al., 2016). Intel official forum even reported examples of $10\times$ to $350\times$ overheads (Feng, 2017). Moreover, based on our experiments, Linux's paging introduce up to runtime $24\times$ on matrix multiplications.

Independent of our work, a few recent[3] proposals also rely on TEE (Hunt et al., 2018; Tople et al., 2018) or TEE and GPU (Tramèr & Boneh, 2019). Chiron (Hunt et al., 2018) assumes the data provider shards training data into n pieces for n enclaves, such that each shard fits in enclave memory. The authors left the policy for managing insufficient enclave memory as future work. Most importantly, Chiron requires new SGX features that are not available yet.

Volos et al. (2018) proposed Graviton, an architecture for supporting TEE on GPU with the help of SGX. As a general purpose TEE, it supports the neural network computation with near-native performance compared to untrusted GPU. However, they assume that an attacker cannot physically steal information from the GPU cores, which is questionable because GPU cores, unlike SGX, are not designed for trusted operation and their security is not well examined.

Tramèr & Boneh (2019) recently proposed Slalom for verifiable and private inference using a trusted enclave which also outsources some computation to a GPU. Their approach heavily relies on the assumption that the server knows the model's parameters. It is thus not applicable to privacy-preserving training. Privado (Tople et al., 2018) allows a model owner to outsource privacy-preserving DNN inference to an SGX-enabled cloud server. It guarantees that even a powerful cloud who sees the SGX enclave memory access pattern does not learn model parameters or the user query. Compare with our solution, Privado does not handle training phase, nor does it leverage untrusted hardware like GPU for acceleration.

Bahmani et al. (2017) proposed an SGX-based framework for general-purpose secure multi-party computation. In general, our work can be viewed as a special protocol instance under their frame-

[3]Our first draft was completed in early '18 while Volos et al. (2018) is officially published in Oct. '18. To the best of our knowledge, they are not yet published after peer-reviews.

work, but we provided additional important contribution that we use untrusted GPU to further accelerate computations.

Kunkel et al. (2019) also propose TensorSCONE to port another popular DNN framework TensorFlow to SCONE. Our basic approach is similar to this framework, and we provide our implementation to public for benchmarking.

Orenbach et al. (2017) proposes Eleos, a memory handling mechanism to reduce performance overhead due to SGX's memory page fault. Its main idea is to prevent, when page fault happens, exiting enclaves because which is an expensive instruction. The experimental results shows that it can reduce the paging overhead by $5\times$. And its successive work CoSMiX (Orenbach et al., 2019) shows that the paging overhead can be further reduced to $1.3 - 2.4\times$. We assume Goten and CaffeSCONE employ this memory handling mechanism to handle paging, and we simulate the performance that does not affected by paging by using simulation mode form Intel SGX SDK.

B.3 Differential Privacy

Another line of research focuses on achieving differential privacy (Dwork, 2006; Dwork et al., 2006a;b). Abadi et al. (2016b) propose a differentially private stochastic gradient descent algorithm for deep learning. Shokri & Shmatikov (2015) propose collaborative learning, in which data owners jointly train a deep neural network by exchanging differentially private gradients through a parameter server instead of directly sharing local training data. Although Shokri & Shmatikov (2015) makes it hard to tell whether a specific record exists in the victim's private training set, it does not prevent an adversary from learning macro-feature of the training set. Phong et al. (2018) showed that the parameter server in Shokri & Shmatikov (2015) can extract information about the training set, and proposed to use additive HE to eliminate the leakage during training.

C Security Analysis

C.1 Protection Scope

From the perspective of the querier, no one else can learn the prediction query and the corresponding result. For the model, the most valuable information includes the parameter of the neural network (*e.g.*, weights and bias of convolutional filters and fully-connected layer), the accuracy according to the training dataset, and the intermediate results. These explicit parameters of the model would not be known by any data-provider and server (with protection against side-channel attacks described shortly afterwards).

We aim for a practical framework instead of a perfectly leakage-free solution. Following the literature (Juvekar et al., 2018; Mohassel & Zhang, 2017; Liu et al., 2017; Gilad-Bachrach et al., 2016), we do not protect the hyper-parameters such as the learning rate, the number of layers, the size of each layer *etc.* These could be inferred by the querier by timing the interaction with the server or by the server from the memory access pattern. One may hide these by adding dummy storage and computation, which is ought to be inefficient.

Side-channel leakage is also out of our protection scope. Specifically, the access pattern in cacheline may reveal information about the data (Ohrimenko et al., 2016). In our case, max-pooling layers and the argmax function in the output layer would be exploitable for their branching depending on the intermediate results. Yet, the existing defence (Ohrimenko et al., 2016) can be easily employed by changing the assembly code of $\max()$ in the enclave, and the computation overhead is less than 2% (Ohrimenko et al., 2016).

Model extraction attacks (Tramèr et al., 2016; Fredrikson et al., 2015) can be launched in a blackbox environment, namely, the attacker knows nothing about the model parameters and its architecture but can query the model, whereby he/she duplicates the functionality of the model. We can easily employ two effective mitigations. First, the training data providers can limit the query rate or set up a query quota by consensus. Second, we only return the labels of evaluation results, instead of the confidence values (the values of the output vector) since it is the main attribute being abused by the attackers (Tramèr et al., 2016).

C.2 Operations inside Enclaves

With the use of SGX, the security guarantee is easy to see. In our construction, the data provided by data providers are either stored in the enclave or sealed on server storage. When data is stored inside the enclave, by the security guarantee of SGX, no other party is able to gain any information. When data is stored outside the enclave, we seal the data by an authenticated encryption (AES-GCM) (Costan & Devadas, 2016), which protects the confidentiality and integrity of a sealed block. We also authenticate the meta-data, in particular, the identity and the number of executions of the block, which disallows arbitrary manipulation of the input data by mix-and-match.

Apart from storage, we also perform execution over the data. In our framework, all executions are data-independent — the executions of neural networks have no branching dependent on the data or models parameters. We analysis our implementation to be data-oblivious using PinTool[], a tool for analysis execution trace, to make sure the trace is the same given model parameter, training data, and prediction queries. The execution view observed by other parties can thus be simulated by without the actual data.

C.2.1 Data-oblivious Operations

The host of an enclave can observe the memory access pattern, even in L2 cache level (Brasser et al., 2017). Hence, we need to ensure algorithms running in enclaves are data-oblivious, meaning that the trace of executed cpu instruction should be the same even given different input data.

Functions involves branching, *e.g.* max, min, may arouse concern on data-oblivious because some optimization of compilers may skip the write instruction if the computed value is equal the original value. For example, the write in $y = \mathsf{max}(y, 0)$ may be skipped if y is indeed large then 0.

Fortunately, we can always use vectorization techniques to avoid such situations. With vectorization techniques, *e.g.*, SSE and AVX, the vectorized read and write instructions will not be skipped since they are atomic and hence no branch depending the data value. Even better, such vectorization techniques are usually automatically employed by common compilers, *e.g.* GCC, with proper flags, *e.g.*, `-march=native`. All we need to do is manually inspecting compiled assemble code or using trace analysis tools, *e.g.* PinTool (Luk et al., 2005), for automatic verification.

C.3 Outsourcing to GPUs

The only cryptographic primitive we used in the outsourcing protocol is additive secret sharing, which is commonly used in the non-colluding server setting (Wang et al., 2014; Mohassel & Zhang, 2017) for privacy-preserving machine learning and also a standard practice in the bigger context of secure multi-party computation (Hohenberger & Lysyanskaya, 2005; Chow et al., 2009; Demmler et al., 2015). Its confidentiality holds in the strong information-theoretic sense against any adversary without enough shares. This fits with the non-colluding server setting well.

Here, we prove that our modified triplet multiplication is secure, namely, none of the server S_0, S_1, and S_2 can gain any information of the contents of a, b, or $c = a \otimes c$ (the servers can learn their dimensions). Due to the non-colluding assumption, we only need to prove that the knowledge of each individual server can be reduced to their counterpart the original protocol.

For S_2, it knows u and v, which are random tensors/matrices and contain no information about a, b, or c. Also, $z = u \otimes v$ derived from u and v contains no extra information.

Speaking at high-level, the extra knowledge of S_0 and S_1 leaks no meaningful information because it is all one-time padding.

Comparing the original protocol described in Section A.4 with our protocol described in Figure 2. in our protocol, S_0 has extra knowledge $\langle z \rangle_1$, $c_1 + K_{1\to 0}$, and $K_{0\to 1}$. Now, we apply the game-hopping technique to prove that our scheme is reducible to the original protocol. Firstly, since S_0 does not know $\langle z \rangle_0$ in our protocol, we can replace $\langle z \rangle_1$ by $\langle z \rangle_0$. Then, since S_0 also does not $K_{1\to 0}$, $c_i + K_{1\to 0}$ can also be replaced by a random matrix/tensor. Likewise, $K_{0\to 1}$ is just another random matrix/tensor so it can be replaced trivially. Now, S_0 has the view of S_0 in the original protocol plus two random matrices/tensors.

```
net = nn.Sequential(
  nn.Conv2d(3, 64, 3, padding=1),
  nn.BatchNorm2d(64), nn.relu(),
  nn.MaxPool2d(2, 2),
  nn.Conv2d(64, 128, 3, padding=1),
  nn.BatchNorm2d(128), nn.relu(),
  nn.MaxPool2d(2, 2),
  nn.Conv2d(128, 256, 3, padding=1),
  nn.BatchNorm2d(256), nn.relu(),
  nn.Conv2d(256, 256, 3, padding=1),
  nn.BatchNorm2d(256), nn.relu(),
  nn.MaxPool2d(2, 2),
  nn.Conv2d(256, 512, 3, padding=1),
  nn.BatchNorm2d(512), nn.relu(),
  nn.Conv2d(512, 512, 3, padding=1),
  nn.BatchNorm2d(512), nn.relu(),
  nn.MaxPool2d(2, 2),
  nn.Conv2d(512, 512, 3, padding=1),
  nn.BatchNorm2d(512), nn.relu(),
  nn.Conv2d(512, 512, 3, padding=1),
  nn.BatchNorm2d(512), nn.relu(),
  nn.MaxPool2d(2, 2),
  nn.Linear(512, 512),
  nn.BatchNorm1d(512), nn.relu(),
  nn.Linear(512, 512),
  nn.BatchNorm1d(512), nn.relu(),
  nn.Linear(512, 10)
)
```

Figure 6: The Architecture of VGG11

Likewise, S_1 has extra knowledge of $c_0 + K_{0\to 1}$ and $K_{1\to 0}$. Applying the same principles for analyzing S_0, it can be reduced to S_1 in the original protocol.