

DELPHI: A Cryptographic Inference Service for Neural Networks

Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa, *UC Berkeley*

https://www.usenix.org/conference/usenixsecurity20/presentation/mishra

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

DELPHI: A Cryptographic Inference Service for Neural Networks

Pratyush Mishra Ryan Lehmkuhl Akshayaram Srinivasan
Wenting Zheng Raluca Ada Popa

UC Berkeley

Abstract

Many companies provide neural network prediction services to users for a wide range of applications. However, current prediction systems compromise one party's privacy: either the user has to send sensitive inputs to the service provider for classification, or the service provider must store its proprietary neural networks on the user's device. The former harms the personal privacy of the user, while the latter reveals the service provider's proprietary model.

We design, implement, and evaluate DELPHI, a secure prediction system that allows two parties to execute neural network inference without revealing either party's data. DELPHI approaches the problem by simultaneously co-designing cryptography and machine learning. We first design a hybrid cryptographic protocol that improves upon the communication and computation costs over prior work. Second, we develop a planner that automatically generates neural network architecture configurations that navigate the performance-accuracy trade-offs of our hybrid protocol. Together, these techniques allow us to achieve a 22× improvement in online prediction latency compared to the state-of-the-art prior work.

1 Introduction

Recent advances in machine learning have driven increasing deployment of neural network inference in popular applications like voice assistants [Bar18] and image classification [Liu+17b]. However, the use of inference in many such applications raises privacy concerns. For example, home monitoring systems (HMS) such as Kuna [Kun] and Wyze [Wyz] use proprietary neural networks to classify objects in video streams of users' homes such as cars parked near the user's house, or faces of visitors to the house. These models are core to these companies' business and are expensive to train.

To make use of these models, either the user has to upload their streams to the servers of the HMS (which then evaluate the model over the stream), or the HMS has to store its model on the user's monitoring device (which then performs the classification). Both of these approaches are unsatisfactory: the first requires users to upload video streams containing sensitive information about their daily activities to another party, while the second requires the HMS to store its model on every device, thus allowing users and competitors to steal the proprietary model.

To alleviate these privacy concerns, a number of recent works have proposed protocols for *cryptographic prediction* over (convolutional) neural networks [Gil+16; Moh+17; Liu+17a; Juv+18] by utilizing specialized secure multi-party computation (MPC) [Yao86; Gol+87]. At a high level, these protocols proceed by encrypting the user's input and the service provider's neural network, and then tailor techniques for computing over encrypted data (like homomorphic encryption or secret sharing) to run inference over the user's input. At the end of the protocol execution, the intended party(-ies) learn the inference result; neither party learns anything else about the other's input. Fig. 1 illustrates this protocol flow.

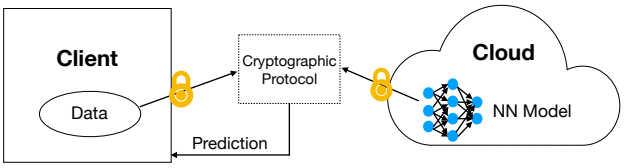


Figure 1: Cryptographic neural network inference. The lock indicates data provided in encrypted form.

Unfortunately, these cryptographic prediction protocols are still unsuitable for deployment in real world applications as they require the use of heavy cryptographic tools during the online execution. These tools are computationally intensive and often require a large amount of communication between the user and the service provider. Furthermore, this cost grows with the complexity of the model, making these protocols unsuitable for use with state-of-the-art neural network architectures used in practice today. For example, using a state-of-the-art protocol like GAZELLE [Juv+18] to perform inference for state-of-the-art deep neural networks like ResNet-32 [He+16] requires ∼ 82 seconds and results in over 560 MB communication.

Our contribution. In this paper, we present DELPHI, a cryptographic prediction system for realistic neural network architectures. DELPHI achieves its performance via a careful co-design of cryptography and machine learning. DELPHI contributes a novel hybrid cryptographic prediction protocol, as well as a planner that can adjust the machine learning algorithm to take advantage of the performance-accuracy trade-offs of our protocol. Our techniques enable us to perform cryptographic prediction on more realistic network architectures than those considered in prior work. For example, using DELPHI for cryptographic prediction on ResNet-32 requires just 3.8 seconds and 60 MB communication in the online phase, improving upon GAZELLE by 22× and 9× respectively.

1.1 Techniques

We now describe at a high level the techniques underlying DELPHI's excellent performance.

Performance goals. Modern convolutional neural networks consist of a number of layers, each of which contains one sub-layer for *linear* operations, and one sub-layer for *non-linear* operations. Common linear operations include convolutions, matrix multiplication, and average pooling. Non-linear operations include activation functions such as the popular ReLU (Rectified Linear Unit) function.

Achieving cryptographic prediction for realistic neural networks thus entails (a) constructing efficient subprotocols for evaluating linear and non-linear layers, and (b) linking the results of these subprotocols with each other.

Prior work. Almost all prior protocols for cryptographic prediction utilize heavyweight cryptographic tools to implement these subprotocols, which results in computation and communication costs that are much higher than the equivalent plaintext costs. Even worse, many protocols utilize these tools during the latency-sensitive *online phase* of the protocol, i.e., when the user acquires their input and wishes to obtain a classification for it. (This is opposed to the less latency-sensitive *preprocessing phase* that occurs before the user's input becomes available).

For example, the online phase of the state-of-the-art GAZELLE protocol uses heavy cryptography like linearly homomorphic encryption and garbled circuits. As we show in Section 7.4, this results in heavy preprocessing *and* online costs: for the popular network architecture ResNet-32 trained over CIFAR-100, GAZELLE requires ~ 158 seconds and 8 GB of communication during the preprocessing phase, and ~ 50 seconds and 5 GB of communication during the preprocessing phase, and ~ 82 seconds and 600 MB of communication during the online phase.

1.1.1 DELPHI's protocol

To achieve good performance on realistic neural networks, DELPHI builds upon techniques from GAZELLE to develop new protocols for evaluating linear and non-linear layers that minimize the use of heavy cryptographic tools, and thus minimizes communication and computation costs in the preprocessing and online phases. We begin with a short overview of GAZELLE's protocol as it is the basis for DELPHI's protocols.

Starting point: GAZELLE. GAZELLE [Juv+18] is a state-of-the-art cryptographic prediction system for convolutional neural networks. GAZELLE computes linear layers using an optimized *linearly-homomorphic encryption* (LHE) scheme [Elg85; Pai99; Reg09; Fan+12] that enables one to perform linear operations directly on ciphertexts. To compute non-linear layers, GAZELLE uses *garbled circuits* [Yao86] to compute the bitwise operations required by ReLU. Finally, because each layer in a neural network consists of alternating linear and non-linear layers, GAZELLE also describes how to efficiently switch back-and-forth between the two aforementioned primitives via a technique based on additive secret sharing.

As noted above, GAZELLE's use of heavy cryptography in the online phase leads to efficiency and communication overheads. To reduce these overheads, we proceed as follows.

Reducing the cost of linear operations. To reduce the online cost of computing the linear operations, we adapt GAZELLE to move the heavy cryptographic operations over LHE ciphertexts to the preprocessing phase. Our key insight is that the service provider's input $\mathbf{M}$ to the linear layer (i.e. the model weights for that layer) is known before user's input is available, and so we can use LHE to create secret shares of $\mathbf{M}$ during preprocessing. Later, when the user's input becomes available in the online phase, all linear operations can be performed directly over secret-shared data without invoking heavy cryptographic tools like LHE, and without requiring interactions to perform matrix-vector multiplications.

The benefits of this technique are two-fold. First, the online phase only requires transmitting secret shares instead of ciphertexts, which immediately results in an $8\times$ reduction in online communication for linear layers. Second, since the online phase only performs computations over elements of prime fields, and since our system uses concretely small 32-bit primes for this purpose, our system can take advantage of state-of-the-art CPU and GPU libraries for computing linear layers; see Section 7.2 and Remark 4.2 for details.

Reducing the cost of non-linear operations. While the above technique already significantly reduces computation time and communication cost, the primary bottleneck for both remains the cost of evaluating garbled circuits for the ReLU activation function. To minimize this cost, we use an alternate approach [Gil+16; Liu+17a; Moh+17; Cho+18] that is better suited to our setting of computing over finite field elements: computing polynomials. In more detail, DELPHI replaces ReLU activations with polynomial (specifically, quadratic) approximations. These can be computed securely and efficiently via standard protocols [Bea95].

Because these protocols only require communicating a small constant number of field elements per multiplication, using quadratic approximations significantly reduces the communication overhead per activation, without introducing additional rounds of communication. Similarly, since the underlying multiplication protocol only requires a few cheap finite field operations, the computation cost is also reduced by several orders of magnitude. Concretely, the online communication and computation costs of securely computing quadratic approximations are $192\times$ and $10000\times$ smaller (respectively) than the corresponding costs for garbled circuits.

However, this performance improvement comes at the cost of accuracy and trainability of the underlying neural network. Prior work has already established that quadratic approximations provide good accuracy in some settings [Moh+17; Liu+17a; Gho+17; Cho+18]. At the same time, both prior

work [Moh+17] and our own experiments indicate that in many settings simply replacing ReLU activations with quadratic approximations results in severely degraded accuracy, and can increase training time by orders of magnitude (if training converges at all). To overcome this, we develop a *hybrid cryptographic protocol* that uses ReLUs *and* quadratic approximations to achieve good accuracy *and* good efficiency.

Planning an efficient usage of the hybrid cryptographic protocol. It turns out that it is not straightforward to determine *which* ReLU activations should be replaced with quadratic approximations. Indeed, as we explain in Section 5, simply replacing arbitrary ReLU activations with quadratic approximations can degrade the accuracy of the resulting network, and can even cause the network to fail to train.

So, to find an appropriate placement or *network configuration*, we design a planner that automatically discovers which ReLUs to replace with quadratic approximations so as to maximize the number of approximations used while still ensuring that accuracy remains above a specified threshold.

The insight behind our planner is to adapt techniques for *neural architecture search* (NAS) and *hyperparameter optimization* (see [Els+19; Wis+19] for in-depth surveys of these areas) to our setting. Namely, we adapt these techniques to discover which layers to approximate within a given neural network architecture, and to optimize the hyperparameters for the discovered network. See Section 5 for details.

The overall system. DELPHI combines the above insights into a cohesive system that service providers can use to automatically generate cryptographic prediction protocols meeting performance and accuracy criteria specified by the provider. In more detail, the service provider invokes DELPHI's planner with acceptable accuracy and performance thresholds. The planner outputs an optimized architecture that meets this goal, which DELPHI then uses to instantiate a concrete cryptographic prediction protocol that utilizes our cryptographic techniques from above.

This co-design of cryptography and machine learning enables DELPHI to efficiently provide cryptographic prediction for networks deeper than any considered in prior work. For example, in Section 7 we show that using DELPHI to provide inference for the popular ResNet-32 architecture requires only 60 MB communication and 3.8 seconds.

2 System overview

2.1 System setup

There are two parties in the system setup: the client and the service provider (or server). In the plaintext version of our system, the service provider provides prediction as a service using its internal models via an API. The client uses this API to run prediction on its own data by transferring its data to the service provider. The service provider runs prediction using the appropriate neural network, then sends the prediction result back to the client. In DELPHI, the two parties execute a secure prediction together by providing their own inputs. The service provider's input is the neural network, while the client's input is its private input used for prediction.

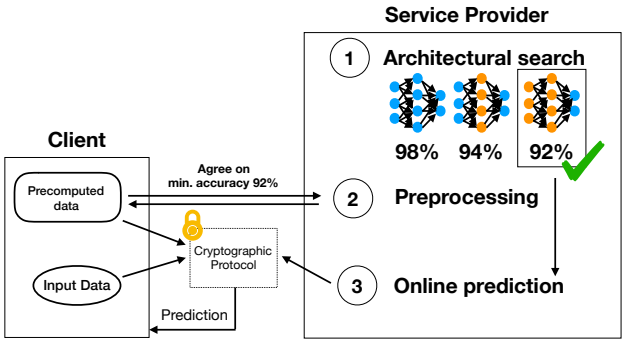


Figure 2: DELPHI's architecture. Orange layers represent quadratic approximations while blue ones represent ReLUs.

2.2 Threat model

DELPHI's threat model is similar to that of prior secure prediction works such as GAZELLE [Juv+18] and MiniONN [Liu+17a]. More specifically, DELPHI is designed for the two-party semi-honest setting, where only one of the parties is corrupted by an adversary. Furthermore, this adversary never deviates from the protocol, but it will try to learn information about the other parties' private inputs from the messages it receives.

2.3 Privacy goals

DELPHI's goal is to enable the client to learn only two pieces of information: the architecture of the neural network, and the result of the inference; all other information about the client's private inputs and the parameters of the server's neural network model should be hidden. Concretely, we aim to achieve a strong simulation-based definition of security; see Definition 4.1.

Like all prior work, DELPHI does not hide information about the architecture of the network, such as the dimensions and type of each layer in the network. For prior work, this is usually not an issue because the architecture is independent of the training data. However, because DELPHI's planner uses training data to optimally place quadratic approximations, revealing the network architecture reveals some information about the data. Concretely, in optimizing an ℓ-layer network, the planner makes ℓ binary choices, thus reveals at most ℓ bits of information about the training data. Because ℓ is concretely small for actual networks (for example, $\ell = 32$ for ResNet32), this leakage is negligible. This leakage can be further mitigated by using differentially private training algorithms [Sho+15; Aba+16]

DELPHI, like most prior systems for cryptographic prediction, does not hide information that is revealed by the result of the prediction. In our opinion, protecting against attacks that exploit this leakage is a complementary problem to that

solved by DELPHI. Indeed, such attacks have been successfully carried out even against systems that "perfectly" hide the model parameters by requiring the client to upload its input to the server [Fre+14; Ate+15; Fre+15; Wu+16b; Tra+16]. Furthermore, popular mitigations for these attacks, such as differential privacy, can be combined with DELPHI's protocol. We discuss these attacks and possible mitigations in more detail in Section 8.2.

2.4 System architecture and workflow

DELPHI's architecture consists of two components: a hybrid cryptographic protocol for evaluating neural networks, and a neural network configuration planner that optimizes a given neural network for use with our protocol. Below we provide an overview of these components, and then demonstrate how one would use these in practice by describing an end-to-end workflow for cryptographic prediction in home monitoring systems (HMS).

Hybrid cryptographic protocol. DELPHI's protocol for cryptographic prediction consists of two phases: an offline preprocessing phase, and an online inference phase. The offline preprocessing phase is independent of the client's input (which regularly changes), but assumes that the server's model is static; if this model changes, then both parties would have to re-run the preprocessing phase. After preprocesing, during the online inference phase, the client provides its input to our specialized secure two-party computation protocol, and eventually learns the inference result. We note that our protocol provides two different methods of evaluating non-linear layers: the first offers better accuracy at the cost of worse offline and online efficiency, while the other degrades accuracy, but offers much improved offline and online efficiency.

Planner. To help service providers navigate the trade off between performance and accuracy offered by these two complementary methods to evaluate non-linear layers, DELPHI adopts a principled approach by designing a *planner* that generates neural networks that mix these two methods to maximize efficiency while still achieving the accuracy desired by the service provider. Our planner applies neural architecture search (NAS) to the cryptographic setting in a novel way in order to automatically discover the right architectures.

Example 2.1 (HMS workflow)**.** As explained in Section 1, a home monitoring system (HMS) enables users to surveil activity inside and outside their houses. Recent HMSes [Kun; Wyz] use neural networks to decide whether a given activity is malicious or not. If it is, they alert the user. In this setting privacy is important for both the user and the HMS provider, which makes DELPHI an ideal fit. To use DELPHI to provide strong privacy, the HMS provider proceeds as follows.

The HMS provider first invokes DELPHI's planner to optimize its baseline all-ReLU neural network model. Then, during the HMS device's idle periods, the device and the HMS server run the preprocessing phase for this model. If the device detects suspicious activity locally, it can run the online inference phase to obtain a classification. On the basis of this result, it can decide whether to alert the user or not.

Remark 2.2 (applications suitable for use with DELPHI)**.** Example 2.1 indicates that DELPHI is best suited for applications where there is ample computational power available for preprocessing, and where inference is latency-sensitive, but is not performed frequently enough to deplete the reserve of preprocessed material. Other examples of such applications include image classification in systems like Google Lens [Goo].

3 Cryptographic primitives

In this section, we provide a high-level description of the cryptographic building blocks used in DELPHI; this high-level description suffices to understand our protocols. We provide formal definitions of security properties in Appendix A, and only provide high level intuitions here.

Garbled circuits. Garbled circuits (GC), introduced in the seminal work of Yao [Yao86], are a method of encoding a boolean circuit C and its input x such that, given the encoded circuit and the encoded input, an evaluator can use a special evaluation procedure to obtain the output $C(x)$ while ensuring that the evaluator learns nothing else about C or x. We now describe this notion in more detail.

A *garbling scheme* [Yao86; Bel+12] is a tuple of algorithms $\mathsf{GS} = (\mathsf{Garble}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{GS.Garble}(C) \to (\tilde{C}, \{\mathsf{label}_{i,0}, \mathsf{label}_{i,1}\}_{i\in[n]})$. On input a boolean circuit C, Garble outputs a *garbled circuit* $\tilde{C}$ and a set of labels $\{\mathsf{label}_{i,0}, \mathsf{label}_{i,1}\}_{i\in[n]}$. Here $\mathsf{label}_{i,b}$ represents assigning the value $b \in \{0,1\}$ to the i-th input label.
- $\mathsf{GS.Eval}(\tilde{C}, \{\mathsf{label}_{i,x_i}\}) \to y$. On input a garbled circuit $\tilde{C}$ and labels $\{\mathsf{label}_{i,x_i}\}$ corresponding to an input $x \in \{0,1\}^n$, Eval outputs a string $y = C(x)$.

We provide a formal definition in Appendix A, and briefly describe here the key properties satisfied by garbling schemes. First, GS must be *complete*: the output of Eval must equal $C(x)$. Second, it must be *private*: given $\tilde{C}$ and $\{\mathsf{label}_{i,x_i}\}$, the evaluator should not learn anything about C or x except the size of $|C|$ (denoted by $1^{|C|}$) and the output $C(x)$.

Linearly homomorphic public-key encryption. A *linearly homomorphic* encryption scheme [Elg85; Pai99] is a public key encryption scheme that additionally supports (only) linearly homomorphic operations on the ciphertexts. To give more details, a linearly homomorphic encryption consists of a tuple of algorithms $\mathsf{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ with the following syntax:

- $\mathsf{HE.KeyGen} \to (\mathsf{pk}, \mathsf{sk})$. HE.KeyGen is a randomized algorithm that outputs a public key pk and a secret key sk.
- $\mathsf{HE.Enc}(\mathsf{pk}, m) \to c$. On input the public key pk and a message m, the encryption algorithm HE.Enc outputs a ciphertext c. The message space is a finite ring $\mathcal{R}$.

- $\mathsf{HE.Dec}(\mathsf{sk}, c) \rightarrow m$. On input the secret key sk and a ciphertext c, the decryption algorithm HE.Dec outputs the message m contained in c.
- $\mathsf{HE.Eval}(\mathsf{pk}, c_1, c_2, L) \rightarrow c'$. On input the public key pk, two ciphertexts c_1, c_2 encrypting messages m_1 and m_2, and a linear function L,[1] HE.Eval outputs a new ciphertext c' encrypting $L(m_1, m_2)$.

Informally, we require HE to satisfy the following properties:

- *Correctness.* HE.Dec, on input sk and a ciphertext $c := \mathsf{HE.Enc}(\mathsf{pk}, m)$, outputs m.
- *Homomorphism.* HE.Dec, on input sk and a ciphertext $c := \mathsf{HE.Eval}(\mathsf{pk}, \mathsf{HE.Enc}(\mathsf{pk}, m_1), \mathsf{HE.Enc}(\mathsf{pk}, m_2), L)$, outputs $L(m_1, m_2)$.
- *Semantic security.* Given a ciphertext c and two messages of the same length, no attacker should be able to tell which message was encrypted in c.
- *Function privacy.* Given a ciphertext c, no attacker can tell what homomorphic operations led to c.

Oblivious transfer. An oblivious transfer protocol [Rab81; Eve+82; Ish+03] is a protocol between two parties, a sender who has as input two messages m_0, m_1, and a receiver who has as input a bit b. At the end of the protocol, the receiver learns m_b. The security requirement states that the sender does not learn anything about bit b and the receiver does not learn anything about the string m_{1-b}.

Additive secret sharing. Given a finite ring $\mathcal{R}$ and an element $x \in \mathcal{R}$, a 2-of-2 *additive secret sharing* of x is a pair $([x]_1, [x]_2) = (x - r, r) \in \mathcal{R}^2$ (so that $x = [x]_1 + [x]_2$) where r is a random element from the ring. Additive secret sharing is perfectly hiding, i.e., given a share $[x]_1$ or $[x]_2$, the value x is perfectly hidden.

Beaver's multiplicative triples. Beaver's multiplication triples [Bea95] generation procedure is a two-party protocol that securely computes the following function. Sample $a, b \leftarrow \mathcal{R}$ and return $[a]_1, [b]_1, [ab]_1$ to the first party and $[a]_2, [b]_2, [ab]_2$ to the second party. In this work, we will generate Beaver's triples using a linearly homomorphic encryption scheme; we provide further details in Appendix A.

Beaver's multiplication procedure. Let P_1 and P_2 be two parties who hold $[x]_1, [y]_1$ and $[x]_2, [y]_2$ respectively where x, y are some ring elements. Additionally, let us assume that P_1 and P_2 also hold a Beaver's multiplication triple, namely, $([a]_1, [b]_1, [ab]_1)$ and $([a]_2, [b]_2, [ab]_2)$ respectively. Beaver's multiplication procedure is a secure protocol such that at the end of the protocol, parties P_1 and P_2 hold an additive secret sharing of xy. We provide details of this protocol in Appendix A but note here that this protocol can be used to securely evaluate any polynomial.

4 Cryptographic protocols

In DELPHI, we introduce a hybrid cryptographic protocol for cryptographic prediction (see Fig. 4). Our protocol makes two key improvements to protocols proposed in prior work like MiniONN [Liu+17a] and GAZELLE [Juv+18]. First, DELPHI splits the protocol into a preprocessing phase and an online phase such that most of the heavy cryptographic computation is performed in the preprocessing phase. Second, DELPHI introduces two different methods of evaluating non-linear functions that provide the users with trade offs between accuracy and performance. The first method uses garbled circuits to evaluate the ReLU activation function, while the second method uses securely evaluates polynomial approximations of the ReLU. The former provides maximum accuracy but is inefficient, while the latter is computationally cheap but lowers accuracy. (We note that below we describe a protocol for evaluating any polynomial approximation, but in the rest of the paper, we restrict ourselves only to quadratic approximations because these are maximally efficient.)

Notation. Let $\mathcal{R}$ be a finite ring. Let $\mathsf{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a linearly homomorphic encryption over the plaintext space $\mathcal{R}$. The server holds a model $\mathbf{M}$ consisting of ℓ layers $\mathbf{M}_1, \ldots, \mathbf{M}_\ell$. The client holds an input vector $\mathbf{x} \in \mathcal{R}^n$.

We now give the formal definition of a cryptographic prediction protocol. Intuitively, the definition guarantees that after the protocol execution, a semi-honest client (i.e., one that follows the specification of the protocol) only learns the architecture of the neural network and the result of the inference; all other information about the parameters of the server's neural network model are hidden. Similarly, a semi-honest server does not learn any information about the client's input, not even the output of the inference.

Definition 4.1. *A protocol Π between a server having as input model parameters $\mathbf{M} = (\mathbf{M}_1, \ldots, \mathbf{M}_\ell)$ and a client having as input a feature vector $\mathbf{x}$ is a* **cryptographic prediction** *protocol if it satisfies the following guarantees.*

- ***Correctness.*** *On every set of model parameters $\mathbf{M}$ that the server holds and every input vector $\mathbf{x}$ of the client, the output of the client at the end of the protocol is the correct prediction $\mathbf{M}(\mathbf{x})$.*
- ***Security:***
  - ***Corrupted client.*** *We require that a corrupted, semi-honest client does not learn anything about the server's network parameters $\mathbf{M}$. Formally, we require the existence of an efficient simulator Sim_C such that $\mathsf{View}_C^\Pi \approx_c \mathsf{Sim}_C(\mathbf{x}, \mathsf{out})$, where View_C^Π denotes the view of the client in the execution of Π (the view includes the client's input, randomness, and the transcript of the protocol), and out denotes the output of the inference.*
  - ***Corrupted server.*** *We require that a corrupted, semi-honest server does not learn anything about the private input $\mathbf{x}$ of the client. Formally, we require the existence of an efficient simulator Sim_S such that $\mathsf{View}_S^\Pi \approx_c \mathsf{Sim}_S(\mathbf{M})$, where View_S^Π denotes the view of the server in the execution of Π.*

[1] L maps (m_1, m_2) to $am_1 + m_2$ for some $a \in \mathcal{R}$.

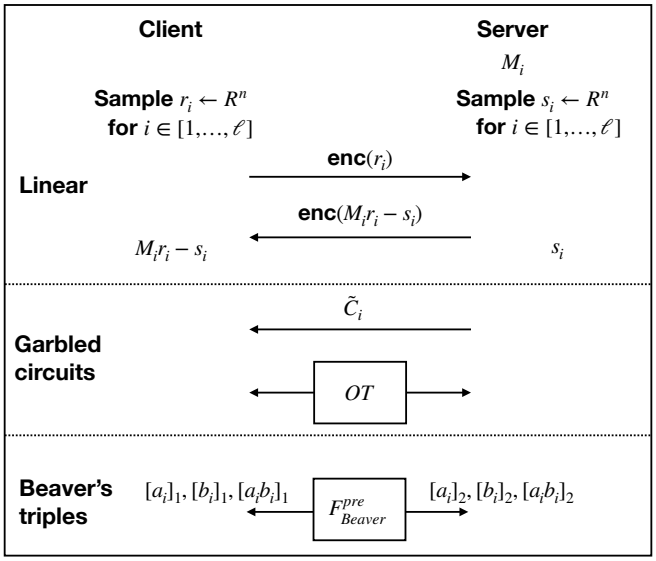


Figure 3: DELPHI's preprocessing phase.

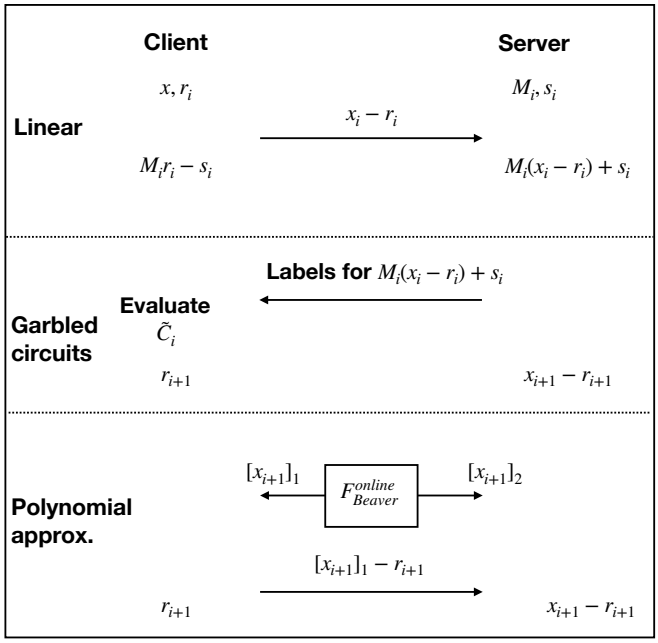


Figure 4: DELPHI's online phase.

The DELPHI protocol proceeds in two phases: the *preprocessing* phase and the *online* phase, and we give the details of both these phases in the subsequent sections.

4.1 Preprocessing phase

During preprocessing, the client and the server pre-compute data that can be used during the online execution. This phase can be executed independent of the input values, i.e., DELPHI can run this phase before either party's input is known.

1. The client runs HE.KeyGen to obtain a public key pk and a secret key sk.
2. For every $i \in [\ell]$, the client and the server choose random masking vectors $\mathbf{r}_i, \mathbf{s}_i \leftarrow \mathcal{R}^n$ respectively.
3. The client sends $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{r}_i)$ to the server. The server computes $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i)$ using the HE.Eval procedure and sends this ciphertext to the client.
4. The client decrypts the above ciphertexts and to obtain $(\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i)$ for each layer. The server holds $\mathbf{s}_i$ for each layer and thus, the client and the server hold an additive secret sharing of $\mathbf{M}_i \mathbf{r}_i$.
5. This step depends on the activation type:
   (a) *ReLU:* The server constructs $\widetilde{C}$ by garbling the circuit C described in Fig. 5. It sends $\widetilde{C}$ to the client and simultaneously, the server and the client exchange labels for the input wires corresponding to $\mathbf{r}_{i+1}$ and $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$ via an Oblivious Transfer (OT).
   (b) *Polynomial approximaitons:* The client and the server run the Beaver's triples generation protocol to generate a number of Beaver's multiplication triples.[2]

[2] The exact number of triples generated depends on the number of layers that have to be approximated using a polynomial.

4.2 Online

The online phase is divided into a two stages: the *setup* and the *layer evaluation*.

4.2.1 Setup

The client on input $\mathbf{x}$, sends $\mathbf{x} - \mathbf{r}_1$ to the server. The server and the client now hold an additive secret sharing of $\mathbf{x}$.

4.2.2 Layer evaluation

At the beginning of the i-th layer, the client holds $\mathbf{r}_i$ and the server holds $\mathbf{x}_i - \mathbf{r}_i$ where $\mathbf{x}_i$ is the vector obtained by evaluating the first $(i-1)$ layers of the neural network on input $\mathbf{x}$ (with $\mathbf{x}_1$ set to $\mathbf{x}$). This invariant will be maintained for each layer. We now describe the protocol for evaluating the i-th layer, which consists of linear functions and activation functions.

Linear layer. The server computes $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$, which ensures that the client and the server an additive secret sharing of $\mathbf{M}_i \mathbf{x}_i$.

Non-linear layer. After the linear functions, the server holds $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ and the client holds $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$. There are two ways of evaluating non-linear layers: garbled circuits for ReLU, or Beaver's multiplication for polynomial approximation:

- **Garbled circuits**
  1. The server sends the garbled labels corresponding to $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$ to the client.
  2. The client evaluates the garbled circuit $\widetilde{C}$ using the above labels as well as the labels obtained via OT (in the offline phase) to obtain a one-time pad ciphertext $\mathrm{OTP}(\mathbf{x}_{i+1} - \mathbf{r}_{i+1})$. It then sends this output to the server.

3. The server uses the one time pad key to obtain $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$.

- **Polynomial approximation**
  1. The client and the server run the Beaver's multiplication procedure to evaluate the polynomial approximating this layer. At the end of the procedure, the client holds $[\mathbf{x}_{i+1}]_1$ and the server holds $[\mathbf{x}_{i+1}]_2$.
  2. The client computes $[x_{i+1}]_1 - \mathbf{r}_{i+1}$ and sends them to the server. The server adds $[\mathbf{x}_{i+1}]_2$ to this value to obtain $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$.

Output layer. The server sends $\mathbf{x}_\ell - \mathbf{r}_\ell$ to the client who adds this with $\mathbf{r}_\ell$ to learn $\mathbf{x}_\ell$.

Hardwired: A random one time pad key.
Input: $\mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i$, $\mathbf{r}_{i+1}$, $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$.
1. Compute $\mathbf{M}_i \cdot \mathbf{x}_i = \mathbf{M}_i(\mathbf{x}_i - \mathbf{r}_i) + \mathbf{s}_i + (\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i)$.
2. Compute $\mathrm{ReLU}(\mathbf{M}_i \cdot \mathbf{x}_i)$ to obtain $\mathbf{x}_{i+1}$.
3. Compute $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$ and output $\mathrm{OTP}(\mathbf{x}_{i+1} - \mathbf{r}_{i+1})$.

Figure 5: A circuit that computes ReLU.

Remark 4.2 (fixed-point arithmetic in finite fields)**.** The discussion so far assumes arithmetic over a finite ring. However, popular implementations of neural network inference perform arithmetic over floating-point numbers. We work around this by using fixed-point representations of floating-point numbers, and embedding this fixed-point arithmetic in our ring arithmetic.

Concretely, our implementation works over the 32-bit prime finite field defined by the prime 2138816513, and uses a 15-bit fixed-point representation. This choice of parameters enables a single multiplication of two fixed-point numbers before the result overflows capacity of the prime field. To prevent values from growing exponentially with the number of multiplications (and thus overflowing), we use a trick from [Moh+17] that allows us to simply truncate the extra LSBs of fixed-point values. This trick works even when the result is secret-shared, albeit at the cost of a 1-bit error.

Similarly to Slalom [Tra+19], our choice of prime field also enables us to losslessly embed our field arithmetic in 64-bit floating point arithmetic. In more detail, 64-bit floating point numbers can represent all integers in the range $2^{-53}, \ldots, 2^{53}$. Because the online phase of our protocol for linear layers requires multiplication of a fixed-point matrix by a secret shared vector, the result is a ~ 45-bit integer, and hence can be represented with full precision in a 64-bit floating point number. This enables our implementation to use state-of-the-art CPU and GPU libraries for linear algebra.

4.3 Security

Theorem 4.3. *Assuming the existence of garbled circuits, linearly homomorphic encryption and secure protocols for Beaver's triples generation and multiplication procedure, the protocol described above is a cryptographic prediction protocol (see Definition 4.1).*

Proof. Below we describe simulators first for the case where the client is corrupted, and then for the case where the server is corrupted. We provide a hybrid argument that relies on these simulators in Appendix B.

4.3.1 Client is corrupted

The simulator Sim, when provided with the client's input $\mathbf{x}$, proceeds as follows:

1. Sim chooses an uniform random tape for the client.
2. In the **offline phase**:
   (a) Sim receives the public key and the ciphertext $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{r}_i)$ from the client. In return, it sends $\mathsf{HE.Enc}(\mathsf{pk}, -\mathbf{s}'_i)$ for a randomly chosen $\mathbf{s}'_i$ from $\mathcal{R}^n$.
   (b) Sim uses the simulator for garbled circuits $\mathsf{Sim}_{\mathsf{GS}}$ and runs it on $1^\lambda, 1^{|C|}$ and sets the output of the circuit to be a random value. $\mathsf{Sim}_{\mathsf{GS}}$ outputs $\widetilde{C}, \{\mathsf{label}_i\}$. For the i-th OT execution, Sim gives the label_i in both slots as input. It sends $\widetilde{C}$ to the client.
   (c) For the secure protocol to generate the Beaver's triples, Sim runs the corresponding simulator for this procedure.
3. **Online phase.** In the preamble phase, Sim receives $\mathbf{x} - \mathbf{r}_1$. It sends $\mathbf{x}$ to the ideal functionality (a semi-honest client uses the same $\mathbf{x}$ as its input) and receives the output $\mathbf{y}$. Sim performs the layer evaluation as follows:
   (a) **Garbled circuits layer.** Sim sends the simulated labels.
   (b) **Polynomial approximation layer.** Sim uses the simulator for the Beaver's multiplication procedure to evaluate the polynomial.
4. **Output layer.** Sim sends $\mathbf{y} - \mathbf{r}_\ell$ to the client.

In Appendix B, we show that the simulated distribution is computationally indistinguishable to the real world distribution using the security of the underlying cryptographic building blocks.

4.3.2 Server is corrupted

The simulator Sim, when provided with the server's input $\mathbf{M}_1, \ldots, \mathbf{M}_{\ell-1}$, proceeds as follows.

1. Sim chooses an uniform random tape for the server.
2. In the **offline phase**:
   (a) Sim chooses a public key pk for a linearly homomorphic encryption scheme. It then sends $\mathsf{HE.Enc}(\mathsf{pk}, \mathbf{0})$ to the server. In return, it receives the homomorphically evaluated ciphertext from the server.
   (b) For every oblivious transfer execution where Sim acts as the receiver, it uses junk input, say 0 as the receiver's choice bit. It receives $\widetilde{C}$ from the server.

(c) For the secure protocol for generating the Beaver's triples, Sim runs the corresponding simulator for this procedure.

3. **Online phase.** In the preamble phase, Sim sends $\mathbf{r}_1$ for an uniformly chosen $\mathbf{r}_1$. Sim performs the layer evaluation step as follows:
   (a) **Garbled circuits layer.** Sim sends a random value back to the server.
   (b) **Polynomial approximation layer.** Sim uses the simulator for the Beaver's multiplication procedure to evaluate the polynomial. At the last round of this step, it sends a random value back to the server.

In Appendix B, we show that the simulated distribution is indistinguishable from the real world distribution using the security of the underlying cryptographic primitives.

5 Planner

DELPHI's planner takes the service provider's neural network model (as well as other constraints) and produces a new neural network architecture that meets the accuracy and efficiency goals of the service provider. At the heart of this planner is an algorithm for neural architecture search (NAS) that enables the service provider to automatically find such network architectures. Below we give a high level overview of this key component and describe how our planner uses it.

Background: neural architecture search. Recently, machine learning research has seen rapid advancement in the area of *neural architecture search* (NAS) [Els+19; Wis+19]. The goal of NAS is to *automatically* discover neural network architectures that best satisfy a set of user-specified constraints. Most NAS algorithms do so by (partially) training a number of different neural networks, evaluating their accuracy, and picking the best-performing ones.

Overview of our planner. DELPHI's planner, when given as input the baseline all-ReLU neural network, operates in two modes. When retraining is either not possible or undesirable (for example if the training data is unavailable or if the provider cannot afford the extra computation required for NAS), the planner operates in the first mode and simply outputs the baseline network. If retraining (and hence NAS) is feasible, then the planner takes as additional inputs the training data, and a constraint on the *minimum* acceptable prediction accuracy t, and then uses NAS to discover a network configuration that maximizes the number of quadratic approximations while still achieving accuracy greater than t. Our planner then further optimizes the hyperparameters of this configuration. In more detail, in this second mode, our planner uses NAS to optimize the following properties of a candidate network configuration given t: (a) the number of quadratic approximations, (b) the placement of these approximations (that is, the layers where ReLUs are replaced with approximations), and (c) training hyperparameters like learning rate and momentum.

The foregoing is a brief description that omits many details. Below, we describe how we solved the challenges that required solving to adapt NAS to this setting (Section 5.1), our concrete choice of NAS algorithm (Section 5.2), and detailed pseudocode for the final algorithm (Fig. 6).

5.1 Adapting NAS for DELPHI's planner

Challenge 1: Training candidate networks. Prior work [Moh+17; Gil+16; Gho+17; Cho+18] and our own experiments indicate that networks that use quadratic approximations are challenging to train and deploy: the quadratic activations cause the underlying gradient descent algorithm to diverge, resulting in poor accuracy. Intuitively, we believe that this behavior is caused by these functions' large and alternating gradients.

To solve this issue, we used the following techniques:

- *Gradient and activation clipping:* During training, we modify our optimizer to use *gradient value clipping*, which helps prevent gradients from exploding [Ben+94]. In particular, we clip the values of all gradients to be less than 2. We furthermore modify our networks to use the ReLU6 activation function [Kri10] that ensures that post-activation values have magnitude at most 6. This keeps errors from compounding during both inference and training.
- *Gradual activation exchange:* Our experiments determined that despite clipping, the gradients were still exploding quickly, especially in deeper networks that contained a higher fraction of approximations. To overcome this, we made use of the following insight: intuitively, ReLU6 and (clipped) quadratic approximations to ReLU should share relatively similar gradients, and so it should be possible to use ReLU6 to initially guide the descent towards a stable region where gradients are smaller, and then to use the approximation's gradients to make fine-grained adjustments within this region.

  We take advantage of this insight by modifying the training process to gradually transform an already-trained all-ReLU6 network into a network with the required number and placement of quadratic approximations. In more detail, our training process expresses each activation as a weighted average of quadratic and ReLU6 activations, i.e., $\mathsf{act}(x) := w_q \cdot \mathsf{quad}(x) + w_r \mathsf{ReLU}(x)$ such that $w_q + w_r = 1$. In the beginning, $w_q = 0$ and $w_r = 1$. Our training algorithm then gradually increases w_q and reduces w_r, so that eventually $w_q = 1$ and $w_r = 0$.

  This technique also improves running times for the NAS as it no longer has to train each candidate network configuration from scratch.

Challenge 2: Efficiently optimizing configurations. Recall from above that our planner aims to optimize the number of quadratic approximations, their placement in the network, and the training hyperparameters. Attempting to optimize all of these variables within a single NAS execution results in a large search space, and finding efficient networks in this

search space takes a correspondingly long time.

To solve this problem, we divided up the monolithic NAS execution into independent runs that are responsible for optimizing different variables. For instance, for an architecture with n non-linear layers, for relevant choices of $m < n$, we first perform NAS to find high-scoring architectures that have m approximation layers, and then perform NAS again to optimize training hyperparameters for these architectures. At the end of this process, our planner outputs a variety of networks with different performance-accuracy trade-offs.

Challenge 3: Prioritizing efficient configurations. Our planner's goal is to choose configurations containing the largest number of approximations in order to maximize efficiency. However, network configurations with large numbers of approximations take longer to train and may be slightly less accurate than networks with fewer approximations. Since the traditional NAS literature focuses on simply maximizing efficiency, using NAS in this default setting results in selecting slower networks over more efficient networks that are just slightly less accurate than the slower ones. To overcome this, we changed the way the NAS assigns "scores" to candidate networks by designing a new scoring function $\mathsf{score}(\cdot)$ which balances prioritizing accuracy and performance. Our experiments from Section 7 indicate that this function enables us to select networks that are both efficient and accurate.

$$\mathsf{score}(N) := \mathrm{acc}(N)\left(1 + \frac{\#\text{quad. activations}}{\#\text{total activations}}\right).$$

5.2 Choosing a NAS algorithm

The discussion so far has been agnostic to the choice of NAS algorithm. In our implementation, we decided to use the popular *population-based training* algorithm [Jad+17] because it was straightforward to customize it for our use case, and because it enjoys a number of optimized implementations (like the one in [Lia+18]).

Population-based training (PBT) [Jad+17] maintains a *population* of candidate neural networks that it trains over a series of time steps. At the end of each time step, it measures the performance of each candidate network via a user-specified scoring function, and replaces the worst-performing candidates with mutated versions of the best-performing ones (the mutation function is specified by the user). At the end of the optimization process, PBT outputs the best-performing candidate network architectures it has found (along with the hyperparameters for training them).

6 System implementation

DELPHI's cryptographic protocols are implemented in Rust and C++. We use the SEAL homomorphic encryption library [Sea] to implement HE, and rely on the `fancy-garbling` library[3] for garbled circuits. To ensure an efficient preprocessing phase, we reimplemented GAZELLE's efficient algorithms for linear layers in SEAL; this may be of independent interest. DELPHI's planner is implemented in Python and uses the scalable PBT [Jad+17] implementation in `Tune` [Lia+18].

$\mathsf{Planner}\left(\begin{array}{ll}\text{all-ReLU6 neural network} & N\\ \text{training data} & D\\ \text{accuracy threshold} & t\end{array}\right)$

1. Let the number of non-linear layers in N be n.
2. Initialize set of output networks $\mathcal{F}$.
3. For i in $\{n/2,\ldots,n\}$:
   (a) Compute the set of best performing models with i quadratic approximation layers: $\mathcal{S}_i \leftarrow \mathsf{PBT}(N, D, \mathsf{score}(\cdot))$.
   (b) Optimize hyperparameters for these models: $\mathcal{S}_i' \leftarrow \mathsf{PBT}(\mathcal{S}_i, D, \mathsf{score}(\cdot))$.
   (c) If for any $N_j \in \mathcal{S}_i'$ the accuracy of N_j is less than t, discard N_j.
   (d) Define N_i to be the network with the maximum score among the remaining networks.
   (e) Set $\mathcal{F} := \mathcal{F} \cup \{N_i\}$.
4. Output $\mathcal{F}$.

Figure 6: Pseudocode for DELPHI's planner.

Remark 6.1 (reimplementing GAZELLE's algorithms)**.** Riazi et al. [Ria+19] note that GAZELLE's implementation does not provide circuit privacy for HE, which can result in leakage of information about linear layers. To remedy this, they recommend using larger parameters that ensure circuit privacy. (The caveat is that these parameters result in worse performance than using GAZELLE's highly optimized parameters.) Because DELPHI uses GAZELLE's algorithms in our preprocessing phase, we attempted to modify GAZELLE's implementation[4] to use the circuit-private parameters. However, this proved to be difficult, and so we decided to reimplement these algorithms in SEAL, which *does* support these parameters.

7 Evaluation

We divide our evaluation into three sections that answer the following questions.

- Section 7.2: How efficient are DELPHI's building blocks?
- Section 7.3: Does DELPHI's planner provide a good balance between efficiency and accuracy for realistic neural networks, such as ResNet-32?
- Section 7.4: What is the latency and communication cost of using DELPHI for serving predictions with such neural networks?

7.1 Evaluation setup

All cryptographic experiments were carried out on AWS `c5.2xlarge` instances possessing an Intel Xeon 8000 series machine CPU at 3.0 GHz with 16 GB of RAM. The client and server were executed on two such instances located in the `us-west-1` (Northern California) and `us-west-2` (Oregon)

[3] https://github.com/GaloisInc/fancy-garbling/

[4] https://github.com/chiraag/gazelle_mpc

regions respectively. The client and server executions used 4 threads each. Machine learning experiments were carried out on various machines with NVIDIA Tesla V100 GPUs. Our machine learning and cryptographic protocol experiments rely on the following datasets and architectures:

1. *CIFAR-10* is a standardized dataset consisting of (32×32) RGB images separated into 10 classes. The training set contains $50,000$ images, while the test set has $10,000$ images. Our experiments use the 7-layer CNN architecture specified in MiniONN [Liu+17a]. Doing so allows us to compare our protocol with prior work.
2. *CIFAR-100* contains the same number of training and test images as CIFAR-10, but divides them up into 100 classes instead of 10. This increased complexity requires a deeper network with more parameters, and so our experiments use the popular ResNet-32 architecture introduced in [He+16]. We note that no prior work on secure inference attempts to evaluate their protocols on difficult datasets like CIFAR-100 or on deep network architectures like ResNet-32.

Whenever we compare DELPHI with GAZELLE, we estimate the cost of GAZELLE's protocols by summing the costs of our re-implementation of the relevant subprotocols for linear and non-linear layers. We do this as there is no end-to-end implementation of GAZELLE's protocol; only the individual subprotocols are implemented.

7.2 Microbenchmarks

We provide microbenchmarks of DELPHI's performance on linear and non-linear layers, comparing both with GAZELLE.

7.2.1 Linear operations

Below we focus on the performance of convolution operations because these comprise the majority of the cost of neural networks' linear operations. The complexity of a convolution is determined by the dimensions of the input and the size and number of convolution kernels, as well as the padding and stride (the latter parameter decides how often the kernel is applied to the input). In Table 1, we evaluate the cost of convolutions used in ResNet-32. The key takeaway is that our online time is over $80\times$ smaller than GAZELLE's, and our online communication is over $150\times$ lower. On the other hand, our preprocessing time and communication are higher than GAZELLE's, but are at most equal to GAZELLE's online time and communication.

Optimized GPU operations. As explained in Remark 4.2, DELPHI's choice of prime field enables DELPHI to use standard GPU libraries for evaluating convolutional layers in the online phase. However, doing so requires copying the layer weights and input into GPU memory, and copying the output back into CPU memory for *every linear layer*. This copying can have substantial overhead. To amortize it, one can batch convolutions over different inputs together. In Table 2, we report the cost of doing so for a batch sizes of 1, 5, and 10. The key takeaway is that, for single convolutions these costs are over 50–$100\times$ lower than the equivalent ones in Table 1, and for batched convolutions, the cost seems to scale sub-linearly with the batch size.

7.2.2 ReLU and quadratic activations

Recall that our protocol for evaluating ReLU activations uses garbled circuits. Our circuit for ReLU follows the design laid out in [Juv+18] with minor additional optimizations. To evaluate quadratic activations, our protocol uses Beaver's multiplication procedure [Bea95], which requires sending one field element from the server to the client and vice versa, and then requires some cheap local field operations from each party. The communication and computation costs for both activations are presented in Table 3.

7.3 DELPHI's planner

To demonstrate the effectiveness of our planner we need to show that (a) quadratic activations are an effective replacement for ReLU activations, and that (b) the networks found by the planner offer better performance than all-ReLU networks. In our experiments below, we use 80% of the training data to train networks in the planner, and the remaining 20% as a validation set. The planner scores candidate networks based on their validation accuracy, but the final reported accuracy is the test set accuracy.

Quadratic activations are effective. We need to show that not only do networks output by our planner achieve good accuracy, but also that the quadratic activations are not redundant. That is, we need to show that the network is not learning to "ignore" quadratic activations. This is a concern because prior work [Mol+17; Liu+18] has shown that modern neural network architectures can be "pruned" to remove extraneous parameters and activations while still maintaining almost the same accuracy.

We show this point by running our planner in two modes. In the first mode, our planner was configured to find performant networks that used quadratic activations, while in the second mode it was configured to find networks that used the identity function instead of quadratic activations, with the intuition that if the quadratic activations were ineffective, then networks that used the identity function instead would perform just as well. The results of these runs for varying number of non-ReLU layers are displayed in Fig. 7 (for CIFAR-10) and in Fig. 8 (for CIFAR-100). Together, these results indicate that the networks output by our planner achieve performance that is comparable to that of the all-ReLU baselines. Furthermore, as the number of non-ReLU layers increase, the best-performing networks that use the identity activation function have much worse accuracy than the equivalent networks that use quadratic activations.

Planned networks perform better. To evaluate the ability of our planner to find networks that offer good performance, we run the planner to produce networks with a varying number (say k) of quadratic layers. We then compare the number of ReLU activations in these networks to that in all-ReLU networks (like those supported by GAZELLE). Fig. 9 illustrates

conv. parameters			system	time (ms)		comm. (MB)	
input $C \times H \times W$	kernel $N \times K \times K$	stride & padding		preproc.	online	preproc.	online
$16 \times 32 \times 32$	$16 \times 3 \times 3$	(1, 1)	**DELPHI**	1236	**15.9**	10.48	**0.065**
			GAZELLE	—	1236	—	10.48
$32 \times 16 \times 16$	$32 \times 3 \times 3$	(1, 1)	**DELPHI**	1262.5	**15.64**	5.24	**0.020**
			GAZELLE	—	1262.5	—	5.24
$64 \times 8 \times 8$	$64 \times 3 \times 3$	(1, 1)	**DELPHI**	2662	**15.6**	5.24	**0.036**
			GAZELLE	—	2662	—	5.24

Table 1: Running time and communication cost of ResNet-32 convolutions in DELPHI.

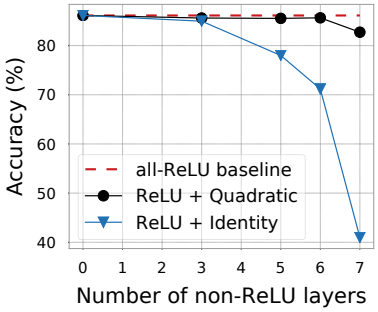


Figure 7: CIFAR-10 accuracy of 7-layer MiniONN networks found by our planner.

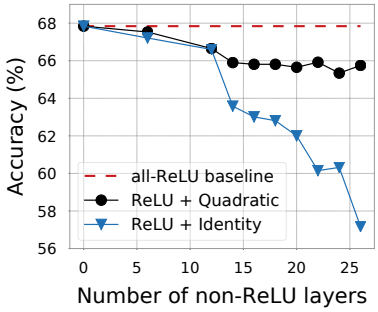


Figure 8: CIFAR-100 accuracy of ResNet32 networks found by our planner.

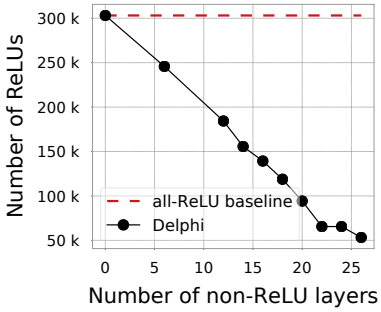


Figure 9: Number of ReLU activations in ResNet32 networks found by our planner.

conv. parameters			time (ms)		
input $C \times H \times W$	kernel $N \times K \times K$	stride & padding	$b = 1$	$b = 5$	$b = 10$
$16 \times 32 \times 32$	$16 \times 3 \times 3$	(1, 1)	0.34	1.07	2.75
$32 \times 16 \times 16$	$32 \times 3 \times 3$	(1, 1)	0.24	0.61	1.10
$64 \times 8 \times 8$	$64 \times 3 \times 3$	(1, 1)	0.24	0.37	0.616

Table 2: Running time and communication cost of ResNet-32 convolutions in DELPHI when run on the GPU across different batch sizes b.

activation function	time (µs)		comm. (kB)	
	preproc.	online	preproc.	online
Quad	6	0.03	0.152	0.008
ReLU	154.9	85.3	17.5	2.048

Table 3: Amortized running time and communication cost of individual ReLU and quadratic activations in DELPHI.

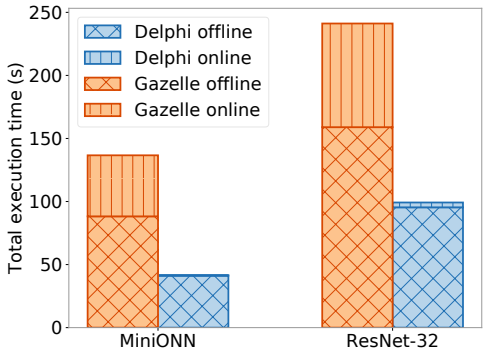


Figure 10: Total execution time on the best planned network (DELPHI) and the all-ReLU baseline (GAZELLE).

this comparison for ResNet32 on CIFAR-100. We observe that the networks found by our planner consistently have fewer activations than the all-ReLU baseline.

7.4 DELPHI's cryptographic protocols

We demonstrate the effectiveness of DELPHI's cryptographic protocol by showing that DELPHI's preprocessing phase and online phase offer significant savings in latency and communication cost over prior work (GAZELLE). Figs. 10 and 11 summarizes this improvement for networks found by our planner; we provide a detailed evaluation next.

Preprocessing phase. Figs. 12a and 13a compare the time required to execute the preprocessing phases of DELPHI and GAZELLE on ResNet32 on CIFAR-100 and the MiniONN architecture on CIFAR-10, respectively. In both cases,

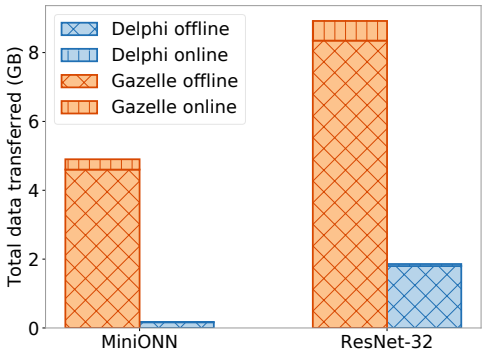


Figure 11: Total communication on the best planned network (DELPHI) and the all-ReLU baseline (GAZELLE).

we observe that, on networks that have a large number of ReLU activations, DELPHI's preprocessing time is larger than GAZELLE's. This is because DELPHI needs to additionally perform preprocessing for each linear layer. However, as the number of approximate activations increases, DELPHI's preprocessing time quickly decreases below that of GAZELLE, because garbling circuits for ReLUs is far more expensive than the preprocessing phase for the approximate activations. A similar trend can be observed for communication costs in Figs. 12c and 13c. Overall, for the most efficient networks output by our planner, DELPHI requires 1.5–2 × less preprocessing time, and 6–40 × less communication.

Online phase. Figs. 12b and 13b compare the time required to execute the online phases of DELPHI and GAZELLE on ResNet32 on CIFAR-100 and the MiniONN architecture on CIFAR-10, respectively. In both cases, we observe that GAZELLE's use of HE for processing linear layers imposes a significant computational cost. Furthermore, as the number of approximate activations increases, the gap between DELPHI and GAZELLE grows larger. A similar trend can be observed for communication costs in Figs. 12d and 13d. Overall, for the most efficient networks output by our planner, DELPHI requires 22–100 × less time to execute its online phase, and 9–40 × less communication.

8 Related work

We first discuss cryptographic techniques for for secure execution of machine learning algorithms in Section 8.1. Then, in Section 8.2, we discuss model inference attacks that recover information about the model from predictions, as well as countermeasures for these attacks. Finally, in Section 8.3, we discuss prior work on neural architecture search.

8.1 Secure machine learning

The problem of secure inference can be solved via generic secure computation techniques like secure two-party (2PC) computation [Yao86; Gol+87], fully homomorphic encryption (FHE) [Gen09], or homomorphic secret sharing (HSS) [Boy+16]. However, the resulting protocols would suffer from terrible communication and computation complexity. For instance, the cost of using 2PC to compute a function grows with the size of the (arithmetic or boolean) circuit for that function. In our setting, the function being computed is the neural network itself. Evaluating the network requires matrix-vector multiplication, and circuits for this operation grow quadratically with the size of the input. Thus using a generic 2PC protocol for secure inference would result in an immediate quadratic blow up in both computation and communication.

Similarly, despite a series of efforts to improve the efficiency of FHE [Bra+11; Gen+11; Fan+12; Hal+18; Hal+19] and HSS [Boy+17], their computational overhead is still large, making them unsuitable for use in our scenario.

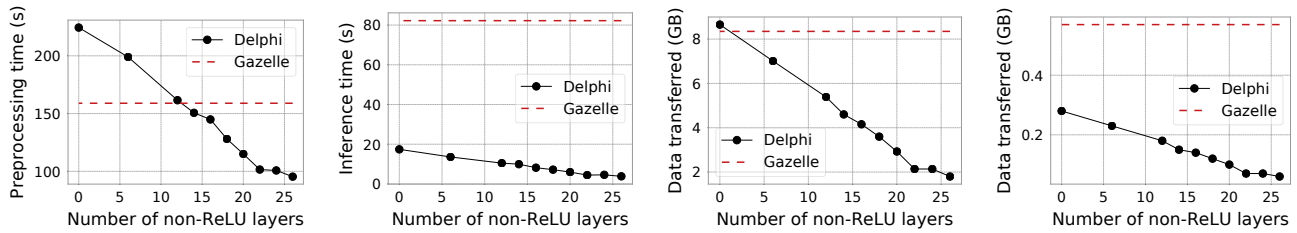
Hence, it seems that it is necessary to design specialized protocols for secure machine learning, and indeed there is a long line of prior work [Du+04; Lau+06; Bar+09; Nik+13a; Nik+13b; Sam+15; Bos+15; Wu+16a; Aon+16; Sch+19] that does exactly this. These works generally fall into two categories: those that focus on secure *training*, and those that focus on secure *inference*. Since secure training is not our focus in this paper, we omit discussing it, and instead focus on prior work on secure inference. Most of these early works focus on simpler machine learning algorithms such as SVMs and linear regression. Designing cryptographic protocols for these simpler algorithms is often more tractable than our setting of inference for neural networks.

Hence, in the rest of this section we discuss prior work that focus on secure inference over neural networks. This work generally falls into the following categories: (a) 2PC-based protocols; (b) FHE-based protocols; (c) TEE-based protocols; and (d) protocols working in a multi-party model.

2PC-based protocols. SecureML [Moh+17] is one of the first systems to focus on the problem of learning and predicting with neural networks securely. However, it relies entirely on generic 2PC protocols to do this, resulting in poor performance on realistic networks. MiniONN [Liu+17a] uses the SPDZ protocol to compute linear layers and polynomial approximation activations. Unlike DELPHI, MiniONN generates multiplicative triples for each multiplication in a linear layer; for a layer with input size n, MiniONN requires n^2 offline and online communication, compared to n for DELPHI.

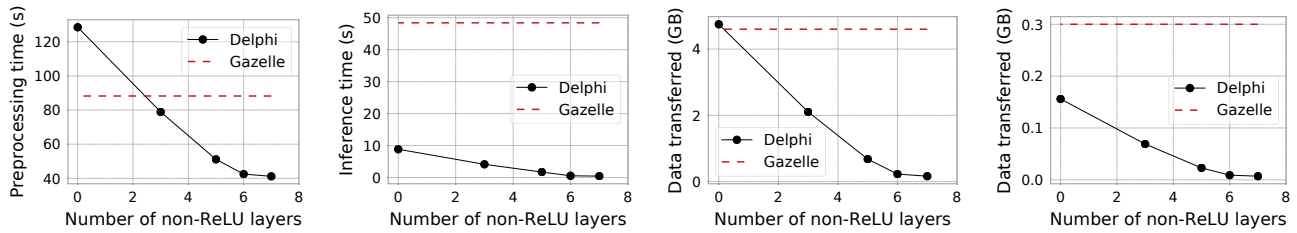
GAZELLE [Juv+18] is the system most similar to ours: it uses an efficient HE-based protocol for linear layers, while using garbled circuits to compute non-linear activations. However, its reliance on heavy cryptographic operations in the online phase results in a protocol that is more expensive than DELPHI's protocol with respect to both computation and communication (see Section 7 for a thorough comparison).

DeepSecure [Rou+18] and XONN [Ria+19] use garbled circuits to provide secure inference for the restricted class of *binarized neural networks* [Cou+15] whose weights are all boolean. This restriction enables these protocols to construct a protocol that uses only a constant number of round trips. DeepSecure additionally prunes the input neural network to reduce the number of activations. Ball et al. [Bal+19] have also recently constructed a protocol for secure inference that



(a) Preprocessing time **(b)** Online time **(c)** Preprocessing communication **(d)** Online communication

Figure 12: Comparison of DELPHI with GAZELLE on the ResNet-32 architecture.



(a) Preprocessing time **(b)** Online time **(c)** Preprocessing communication **(d)** Online communication

Figure 13: Comparison of DELPHI with GAZELLE on the architecture from MiniONN [Liu+17a].

relies on the garbling schemes of [Bal+16]. Unlike XONN and DeepSecure, the protocol of [Bal+19] supports general neural networks. Despite optimizations, each of these works suffers from large concrete costs because each work performs matrix-vector multiplications inside the garbled circuit.

EzPC [Cha+17], on input a high-level description of a program, synthesizes a cryptographic protocol implementing that program. The compiled protocol intelligently uses a mix of arithmetic and boolean 2PC protocols to increase efficiency.

FHE-based protocols. CryptoNets [Gil+16] is the first work that attempts to optimize and tailor FHE schemes for secure inference. Despite optimizations, the limitations of FHE mean that CryptoNets is limited to networks only a few layers deep, and even for these networks it only becomes efficient when processing a batch of inputs. Recent papers [Hes+17; Bru+18; Bou+18; Cho+18; San+18] develop different approaches to optimize the CryptoNets paradigm, but the resulting protocols still require tens of minutes to provide predictions over networks much smaller than the ones we consider here.

CHET [Dat+19] compiles high-level specifications of neural network to FHE-based inference protocols. To efficiently use FHE, CHET must replace *all* ReLUs with polynomial approximations, which harms accuracy for large networks.

TEE-based protocols. There are two approaches for inference using trusted execution enclaves (TEEs): (a) inference via server-side enclaves, where the client uploads their input to the server's enclave, and (b) inference in client-side enclaves, where the client submits queries to a model stored in the client-side enclave.

Slalom and Privado are examples of protocols that rely on server-side enclaves. Slalom [Tra+19], like DELPHI, splits inference into an offline and online phase, and uses additive secret sharing for the online phase. Unlike DELPHI, Slalom uses the Intel SGX hardware enclave [McK+13] to securely compute both the offline and online phases. Privado [Top+18] compiles neural networks into *oblivious* neural networks, meaning that computing the transformed network does not require branching on secret data. They use the oblivious network to perform inference inside Intel SGX enclaves. Slalom's implementation indicates that it does not implement linear or non-linear layers obliviously.

MLCapsule [Han+18] describes a system for performing inference via client-side enclaves. Apple uses a client-side secure enclave to perform fingerprint and face matching to authorize users [App19].

In general, most TEE-based cryptographic inference protocols offer better efficiency than protocols that rely on cryptographic (like DELPHI). This improved efficiency comes at the cost of a weaker threat model that requires trust in hardware vendors and the implementation of the enclave. Furthermore, because the protocol execution occurs in an adversarial environment, any side-channel leakage is more dangerous (since the adversary can carefully manipulate the execution to force this leakage). Indeed, the past few years have seen a number of powerful side-channel attacks [Bra+17; Häh+17; Göt+17; Mog+17; Sch+17; Wan+17; Van+18] against popular enclaves like Intel SGX and ARM TrustZone.

Protocols with more parties. The discussion above focuses

on two-party protocols, because in our opinion secure inference maps naturally to this setting. Nevertheless, a number of works [Ria+18; Wag+18; Tfe; Bar+19] have instead targeted the three-party setting where shares of the model are divided amongst two non-colluding servers, and a client must interact with these servers to obtain their prediction.

8.2 Model leakage from predictions

Prediction API attacks [Ate+15; Fre+15; Wu+16b; Tra+16; Sho+17; Jag+19] aim to learn private information about the server's model or training data given access only to the results of predictions on arbitrary queries.

There is no general defense against prediction API attacks beyond rate limiting and query auditing [Jag+19]. However, there are defenses against specific classes of attacks. For example, one can use differentially private training [Sho+15; Aba+16] to train neural networks that that do not leak sensitive information about the underlying training data.

The guarantees of DELPHI are complementary to those provided by any such mitigations. Indeed, with sufficient effort, these techniques can be integrated into DELPHI to provide even stronger privacy guarantees; we leave this to future work.

8.3 Neural architecture search

Recently, machine learning research has seen rapid advancement in the area of *neural architecture search* (NAS) (see [Els+19; Wis+19] for surveys). The aim of this field is to develop methods to automatically optimize properties of a neural network like accuracy and efficiency by optimizing the hyperparameters of the network. Examples of commonly optimized hyperparameters include the size of convolutional kernels, the number of layers, and parameters of the gradient descent algorithm like learning rate and momentum. In this work, we rely on NAS algorithms only for optimizing the placement of quadratic approximation layers within a network, as ReLU activations were the bottleneck in our system.

Common approaches to neural architecture search include those based on reinforcement-learning [Zop+17], evolutionary algorithms [Yao99; Ber+13], and random search [Ber+12; Jad+17]. DELPHI's planner uses the Population-Based Training algorithm [Jad+17] to perform NAS. PBT can be seen as a hybrid of the evolutionary algorithm and random search approaches.

9 Acknowledgements

We thank Liam Li for the suggestion to use the PBT algorithm to perform NAS, Joey Gonzalez for answering questions about PBT, Robert Nishihara for the suggestion to use ReLU's gradients to guide gradient descent, Chiraag Juvekar for providing the code for GAZELLE, and our shepherd Siddharth Garg and the anonymous reviewers for their invaluable feedback. This work was supported by the NSF CISE Expeditions Award CCF-1730628, as well as gifts from the Sloan Foundation, Bakar and Hellman Fellows Fund, Alibaba, Amazon Web Services, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[Aba+16] M. Abadi, A. Chu, I. J. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. "Deep Learning with Differential Privacy". In: CCS '16.

[Aon+16] Y. Aono, T. Hayashi, L. T. Phong, and L. Wang. "Scalable and Secure Logistic Regression via Homomorphic Encryption". In: CODASPY '16.

[App19] Apple. "iOS Security". https://www.apple.com/business/docs/site/iOS_Security_Guide.pdf.

[Ate+15] G. Ateniese, L. V. Mancini, A. Spognardi, A. Villani, D. Vitali, and G. Felici. "Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers". In: *IJSN* (2015).

[Bal+16] M. Ball, T. Malkin, and M. Rosulek. "Garbling Gadgets for Boolean and Arithmetic Circuits". In: CCS '16.

[Bal+19] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski. "Garbled Neural Networks are Practical". ePrint Report 2019/338.

[Bar+09] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A. Sadeghi, and T. Schneider. "Secure Evaluation of Private Linear Branching Programs with Medical Applications". In: ESORICS '09.

[Bar18] B. Barrett. "The year Alexa grew up". https://www.wired.com/story/amazon-alexa-2018-machine-learning/.

[Bar+19] A. Barak, D. Escudero, A. Dalskov, and M. Keller. "Secure Evaluation of Quantized Neural Networks". ePrint Report 2019/131.

[Bea95] D. Beaver. "Precomputing Oblivious Transfer". In: CRYPTO '95.

[Bel+12] M. Bellare, V. T. Hoang, and P. Rogaway. "Foundations of garbled circuits". In: CCS '12.

[Ben+94] Y. Bengio, P. Y. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE Trans. Neural Networks* (1994).

[Ber+12] J. Bergstra and Y. Bengio. "Random Search for Hyper-Parameter Optimization". In: *JMLR* (2012).

[Ber+13] J. Bergstra, D. Yamins, and D. D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures". In: ICML '13.

[Bos+15] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. “Machine Learning Classification over Encrypted Data”. In: NDSS ’15.

[Bou+18] F. Bourse, M. Minelli, M. Minihold, and P. Paillier. “Fast Homomorphic Evaluation of Deep Discretized Neural Networks”. In: CRYPTO ’18.

[Boy+16] E. Boyle, N. Gilboa, and Y. Ishai. “Function Secret Sharing: Improvements and Extensions”. In: CCS ’16.

[Boy+17] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, and M. Orrú. “Homomorphic Secret Sharing: Optimizations and Applications”. In: CCS ’17.

[Bra+11] Z. Brakerski and V. Vaikuntanathan. “Efficient Fully Homomorphic Encryption from (Standard) LWE”. In: FOCS ’11.

[Bra+17] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: WOOT ’17.

[Bru+18] A. Brutzkus, O. Elisha, and R. Gilad-Bachrach. “Low Latency Privacy Preserving Inference”. ArXiV, cs.CR 1812.10659.

[Cha+17] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. “EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation for Machine Learning”. ePrint Report 2017/1109.

[Cho+18] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei. “Faster CryptoNets: Leveraging Sparsity for Real-World Encrypted Inference”. ArXiV, cs.CR 1811.09953.

[Cou+15] M. Courbariaux, Y. Bengio, and J. David. “BinaryConnect: Training Deep Neural Networks with binary weights during propagations”. In: NeurIPS ’18.

[Dat+19] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. E. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz. “CHET: An optimizing compiler for fully-homomorphic neural-network inferencing”. In: PLDI ’19.

[Du+04] W. Du, Y. S. Han, and S. Chen. “Privacy-Preserving Multivariate Statistical Analysis: Linear Regression and Classification”. In: SDM ’04.

[Elg85] T. Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms”. In: *IEEE Trans. on Inf. Theory* (1985).

[Els+19] T. Elsken, J. H. Metzen, and F. Hutter. “Neural Architecture Search: A Survey”. In: *JMLR* (2019).

[Eve+82] S. Even, O. Goldreich, and A. Lempel. “A Randomized Protocol for Signing Contracts”. In: CRYPTO ’82.

[Fan+12] J. Fan and F. Vercauteren. “Somewhat Practical Fully Homomorphic Encryption”. ePrint Report 2012/144.

[Fre+14] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart. “Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing”. In: USENIX Security ’14.

[Fre+15] M. Fredrikson, S. Jha, and T. Ristenpart. “Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures”. In: CCS ’15.

[Gen09] C. Gentry. “Fully homomorphic encryption using ideal lattices”. In: STOC ’09.

[Gen+11] C. Gentry and S. Halevi. “Implementing Gentry’s Fully-Homomorphic Encryption Scheme”. In: EUROCRYPT ’11.

[Gho+17] Z. Ghodsi, T. Gu, and S. Garg. “SafetyNets: Verifiable Execution of Deep Neural Networks on an Untrusted Cloud”. In: NIPS ’17.

[Gil+16] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: ICML ’16.

[Gol+87] O. Goldreich, S. Micali, and A. Wigderson. “How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority”. In: STOC ’87.

[Goo] Google. “Google Lens”. https://lens.google.com/.

[Göt+17] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. “Cache Attacks on Intel SGX”. In: EUROSEC ’17.

[Häh+17] M. Hähnel, W. Cui, and M. Peinado. “High-Resolution Side Channels for Untrusted Operating Systems”. In: ATC ’2017.

[Hal+18] S. Halevi and V. Shoup. “Faster Homomorphic Linear Transformations in HElib”. In: CRYPTO ’18.

[Hal+19] S. Halevi, Y. Polyakov, and V. Shoup. “An Improved RNS Variant of the BFV Homomorphic Encryption Scheme”. In: CT-RSA ’19.

[Han+18] L. Hanzlik, Y. Zhang, K. Grosse, A. Salem, M. Augustin, M. Backes, and M. Fritz. “MLCapsule: Guarded Offline Deployment of Machine Learning as a Service”. ArXiV, cs.CR 1808.00590.

[He+16] K. He, X. Zhang, S. Ren, and J. Sun. “Deep Residual Learning for Image Recognition”. In: CVPR ’16.

[Hes+17] E. Hesamifard, H. Takabi, and M. Ghasemi. “CryptoDL: Deep Neural Networks over Encrypted Data”. ArXiV, cs.CR 1711.05189.

[Ish+03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. “Extending Oblivious Transfers Efficiently”. In: CRYPTO ’03.

[Jad+17] M. Jaderberg, V. Dalibard, S. Osindero, W. Czarnecki, J. Donahue, A. Razavi, et al. “Population Based Training of Neural Networks”. ArXiV, cs.LG 1711.09846.

[Jag+19] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot. “High-Fidelity Extraction of Neural Network Models”. ArXiV, cs.LG 1909.01838.

[Juv+18] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. “GAZELLE: A Low Latency Framework for Secure Neural Network Inference”. In: USENIX ’18.

[Kri10] A. Krizhevsky. “Convolutional Deep Belief Networks on CIFAR-10”. Unpublished manuscript. http://www.cs.utoronto.ca/~kriz/conv-cifar10-aug2010.pdf.

[Kun] Kuna. “Kuna AI”. https://getkuna.com/blogs/news/2017-05-24-introducing-kuna-ai.

[Lau+06] S. Laur, H. Lipmaa, and T. Mielikäinen. “Cryptographically private support vector machines”. In: KDD ’06.

[Lia+18] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. “Tune: A Research Platform for Distributed Model Selection and Training”. In:

[Liu+17a] J. Liu, M. Juuti, Y. Lu, and N. Asokan. “Oblivious Neural Network Predictions via MiniONN Transformations”. In: CCS ’17.

[Liu+17b] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi. “A survey of deep neural network architectures and their applications”. In: *Neurocomputing* (2017).

[Liu+18] K. Liu, B. Dolan-Gavitt, and S. Garg. “Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks”. In: RAID ’2018.

[McK+13] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. “Innovative instructions and software model for isolated execution”. In: HASP ’13.

[Mog+17] A. Moghimi, G. Irazoqui, and T. Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: CHES ’17.

[Moh+17] P. Mohassel and Y. Zhang. “SecureML: A System for Scalable Privacy-Preserving Machine Learning”. In: IEEE S&P ’17.

[Mol+17] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. “Pruning Convolutional Neural Networks for Resource Efficient Inference”. In: ICLR ’17.

[Nik+13a] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh. “Privacy-preserving matrix factorization”. In: CCS ’13.

[Nik+13b] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. “Privacy-Preserving Ridge Regression on Hundreds of Millions of Records”. In: IEEE S&P ’13.

[Pai99] P. Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: EUROCRYPT ’99.

[Rab81] M. O. Rabin. “How To Exchange Secrets with Oblivious Transfer”. Harvard University Technical Report 81 (TR-81).

[Reg09] O. Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *JACM* (2009).

[Ria+18] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar. “Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications”. In: AsiaCCS ’18.

[Ria+19] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar. “XONN: XNOR-based Oblivious Deep Neural Network Inference”. In: USENIX ’19.

[Rou+18] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. “DeepSecure: Scalable Provably-secure Deep Learning”. In: DAC ’18.

[Sam+15] B. K. Samanthula, Y. Elmehdwi, and W. Jiang. “k-Nearest Neighbor Classification over Semantically Secure Encrypted Relational Data”. In: *IEEE Trans. Knowl. Data Eng.* (2015).

[San+18] A. Sanyal, M. Kusner, A. Gascón, and V. Kanade. “TAPAS: Tricks to Accelerate (encrypted) Prediction As a Service”. In: ICML ’18.

[Sch+17] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: DIMVA ’17.

[Sch+19] P. Schoppmann, A. Gascon, M. Raykova, and B. Pinkas. "Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning". ePrint Report 2019/281.

[Sea] "Microsoft SEAL (release 3.3)". https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[Sho+15] R. Shokri and V. Shmatikov. "Privacy-Preserving Deep Learning". In: CCS '15.

[Sho+17] R. Shokri, M. Stronati, C. Song, and V. Shmatikov. "Membership Inference Attacks Against Machine Learning Models". In: S&P '17.

[Tfe] "TF Encrypted". https://github.com/mortendahl/tf-encrypted.

[Top+18] S. Tople, K. Grover, S. Shinde, R. Bhagwan, and R. Ramjee. "Privado: Practical and Secure DNN Inference". ArXiV, cs.CR 1810.00602.

[Tra+16] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. "Stealing Machine Learning Models via Prediction APIs". In: USENIX Security '16.

[Tra+19] F. Tramer and D. Boneh. "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware". In: ICLR '19.

[Van+18] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: USENIX Security '18.

[Wag+18] S. Wagh, D. Gupta, and N. Chandran. "SecureNN: Efficient and Private Neural Network Training". ePrint Report 2018/442.

[Wan+17] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: CCS '17.

[Wis+19] M. Wistuba, A. Rawat, and T. Pedapati. "A Survey on Neural Architecture Search". ArXiV, cs.LG 1905.01392.

[Wu+16a] D. J. Wu, T. Feng, M. Naehrig, and K. E. Lauter. "Privately Evaluating Decision Trees and Random Forests". In: *PoPETs* (2016).

[Wu+16b] X. Wu, M. Fredrikson, S. Jha, and J. F. Naughton. "A Methodology for Formalizing Model-Inversion Attacks". In: CSF '16.

[Wyz] "Wyze: Contact and Motion Sensors for Your Home". https://www.wyze.com/.

[Yao86] A. C. Yao. "How to Generate and Exchange Secrets (Extended Abstract)". In: FOCS '86.

[Yao99] X. Yao. "Evolving artificial neural networks". In: *Proceedings of the IEEE* (1999).

[Zop+17] B. Zoph and Q. V. Le. "Neural Architecture Search with Reinforcement Learning". In: ICLR '17.

A Security properties of our building blocks

Security for **garbled circuits** requires the existence of a simulator $\mathsf{Sim}_{\mathsf{GS}}$ that, given input $1^\lambda, 1^{|C|}$, and $C(x)$, outputs $\tilde{C}, \{\mathsf{label}_i\}_{i\in[n]}$ such that this output is computationally indistinguishable to $(\tilde{C}, \{\mathsf{label}_{i,x_i}\})$ generated by $\mathsf{GS.Garble}$.

Security for **linearly homomorphic encryption** schemes requires the scheme to satisfy the following properties:

- *Semantic security.* For any two messages m, m', we require $\{\mathsf{pk}, \mathsf{HE.Enc}(\mathsf{pk}, m)\} \approx_c \{\mathsf{pk}, \mathsf{HE.Enc}(\mathsf{pk}, m')\}$, where the two distributions are over the random choice of pk and the random coins of the encryption algorithm.
- *Function privacy.* There exists a simulator $\mathsf{Sim}_{\mathsf{FP}}$ such that for every efficient adversary $\mathcal{A}$, every linear function L, and every pair of messages m_1, m_2, we have that the following distributions are computationally indistinguishable:

$$\left\{ (r, r_1, r_2, c') : \begin{array}{l} (r, r_1, r_2) \leftarrow \{0,1\}^\lambda \\ (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{HE.KeyGen}(1^\lambda; r) \\ c_1 \leftarrow \mathsf{HE.Enc}(\mathsf{pk}, m_1; r_1) \\ c_2 \leftarrow \mathsf{HE.Enc}(\mathsf{pk}, m_2; r_2) \\ c' \leftarrow \mathsf{HE.Eval}(\mathsf{pk}, c_1, c_2, L) \end{array} \right\}$$
$$\approx_c$$
$$\mathsf{Sim}_{\mathsf{FP}}(1^\lambda, m_1, m_2, L(m_1, m_2))$$

B Security proofs

Proof of indistinguishability with corrupted client. We show that the real world distribution is computationally indistinguishable to the simulated distribution via a hybrid argument. In the final simulated distribution, the simulator does not use the weights for the server's model, and so a corrupted client learns nothing beyond the output prediction and the model architecture in the real world.

- Hyb_0: This corresponds to the real world distribution where the server uses its input matrices $\mathbf{M}_1, \ldots, \mathbf{M}_{\ell-1}$.
- Hyb_1: This hybrid involves only a syntactic change. In the output phase, the simulator sends $\mathbf{y} - \mathbf{r}_\ell$ to the client, where $\mathbf{y}$ is the output of the neural network on input $\mathbf{x}$. Additionally, the simulator uses the knowledge of the client's random tape to begin the evaluation of the i-th layer with $\mathbf{x}_i - \mathbf{r}_i$. Since this is a syntactic change, Hyb_1 is distributed identically to Hyb_0.
- Hyb_2: We change the inputs that the server provides to each OT execution where it acts as the sender. Instead of

providing the labels corresponding to 0 and 1 in each OT execution, the server provides $\mathsf{label}_{i,b}$ where b is the input used by the client in that OT execution. Note that in the semi-honest setting, we know b as a result of setting the random tape as well learning the input of the corrupted client. It follows from the sender security of OT that Hyb_2 is indistinguishable from Hyb_1.

- Hyb_3: In this hybrid, for every layer of the neural network that uses garbled circuits, we generate $\widetilde{C}$ using $\mathsf{Sim}_{\mathsf{GS}}$ on input $1^{\lambda}, 1^{|C|}$ and $C(z)$ where z is the input that the client uses to evaluate this circuit (this is again known in the semi-honest setting as a result of setting the random tape and knowing the input). Note that $C(z)$ is an OTP encryption and hence is distributed identically to a random string. It follows from the security of the garbled circuits that Hyb_3 is indistinguishable from Hyb_2.
- Hyb_4: In this hybrid, we generate the multiplication triples in the offline phase using the corresponding simulator for Beaver's protocol. It follows from the simulation security of this protocol that Hyb_4 is indistinguishable from Hyb_3.
- Hyb_5: In this hybrid, for every quadratic approximation layer, we use the simulator for Beaver's multiplication procedure. It again follows from the simulation security that this hybrid is indistinguishable to the previous hybrid. Notice that in this hybrid, the server is no longer using $\mathbf{x}_i - \mathbf{r}_i, \mathbf{s}_i$ as well as the matrix $\mathbf{M}_i$ to evaluate the i-th layer.
- Hyb_6: For every homomorphic evaluation in the offline phase, we use the simulator $\mathsf{Sim}_{\mathsf{FP}}$ for the function privacy of HE. Note that $\mathsf{Sim}_{\mathsf{FP}}$ only requires the output $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$ to generate the homomorphically evaluated ciphertext. It follows from the function privacy of HE that Hyb_6 is computationally indistinguishable from Hyb_5.
- Hyb_7: In this hybrid, we replace the input $-\mathbf{s}'_i$ given to $\mathsf{Sim}_{\mathsf{FP}}$ with randomly sampled $\mathbf{s}'_i$ from $\mathcal{R}^n$ (instead of the true value $\mathbf{M}_i \cdot \mathbf{r}_i - \mathbf{s}_i$). Thus Hyb_7 is distributed identically to Hyb_6 as $\mathbf{s}_i$ is chosen uniformly at random. Finally, we note that Hyb_7 is identically distributed to the simulator's output, completing the proof.

Proof of indistinguishability with corrupted server. We show that the real world distribution is computationally indistinguishable to the simulated distribution via a hybrid argument. In the final simulated distribution, the simulator does not use the user's input, and so a corrupted server learns nothing in the real world.

- Hyb_0: This corresponds to the real world distribution where the client uses its actual input $\mathbf{x}$.
- Hyb_1: This hybrid involves only a syntactic change. For every layer that is evaluated by garbled circuits, instead of evaluating the circuits, we instead send $OTP(\mathbf{x}_{i+1} - \mathbf{r}_{i+1})$ by using our knowledge of $\mathbf{x}$, the matrices $\mathbf{M}_i$, and the random tape of the server. Similarly, in every quadratic approximation layer, we send a share in the final round such that when the server adds it with its own share it gets $\mathbf{x}_{i+1} - \mathbf{r}_{i+1}$. Because this change is only syntactic, Hyb_1 is identical to Hyb_0.
- Hyb_2: In this hybrid, we change the inputs that the client provides to each OT execution where it is acting as the receiver. Instead of providing the actual inputs, it provides some junk inputs, say 0. It follows from the receiver security of the underlying oblivious transfer protocol that Hyb_2 is computationally indistinguishable from Hyb_1.
- Hyb_3: In this hybrid, we generate the multiplication triples in the offline phase using the simulator for Beaver's multiplication protocol. It follows from the simulation security of this protocol that Hyb_4 is indistinguishable from Hyb_2.
- Hyb_4: In this hybrid, for every quadratic approximation layer of the neural network, we use the simulator for the Beaver's multiplication procedure. It follows from simulation security that Hyb_4 is indistinguishable from Hyb_3.
- Hyb_5: In this hybrid, we change the ciphertexts sent by the client in the offline phase. Instead of sending encryptions of $\mathbf{r}_i$, the client sends $\mathsf{HE.Enc}(pk, \mathbf{0})$. It follows from the semantic security of the encryption scheme that Hyb_5 is computationally indistinguishable from Hyb_4.
- Hyb_6: In this hybrid, we make the following changes. For every layer that is evaluated by garbled circuits, we send $OTP(\mathbf{r}_{i+1})$ for a randomly chosen $\mathbf{r}_{i+1}$. Similarly, in every quadratic approximation layer, we send a share in the final round that is chosen uniformly at random. Additionally, in the preamble phase, we send an uniformly chosen value $\mathbf{r}_1$. Hyb_6 is distributed identically to Hyb_5. Finally, note that Hyb_6 is identically distributed to the simulator's output, completing the proof.