2020 IEEE Symposium on Security and Privacy

CRYPTFLOW: Secure TensorFlow Inference

Nishant Kumar*
Microsoft Research
t-niskum@microsoft.com

Mayank Rathee*
Microsoft Research
t-may@microsoft.com

Nishanth Chandran
Microsoft Research
nichandr@microsoft.com

Divya Gupta
Microsoft Research
digup@microsoft.com

Aseem Rastogi
Microsoft Research
aseemr@microsoft.com

Rahul Sharma
Microsoft Research
rahsha@microsoft.com

***Abstract*— We present CRYPTFLOW, a first of its kind system that converts TensorFlow inference code into Secure Multi-party Computation (MPC) protocols at the push of a button. To do this, we build three components. Our first component, Athos, is an end-to-end compiler from TensorFlow to a variety of semi-honest MPC protocols. The second component, Porthos, is an improved semi-honest 3-party protocol that provides significant speedups for TensorFlow like applications. Finally, to provide malicious secure MPC protocols, our third component, Aramis, is a novel technique that uses hardware with integrity guarantees to convert any semi-honest MPC protocol into an MPC protocol that provides malicious security. The malicious security of the protocols output by Aramis relies on integrity of the hardware and semi-honest security of MPC. Moreover, our system matches the inference accuracy of plaintext TensorFlow.**

We experimentally demonstrate the power of our system by showing the secure inference of real-world neural networks such as RESNET50 and DENSENET121 over the ImageNet dataset with running times of about 30 seconds for semi-honest security and under two minutes for malicious security. Prior work in the area of secure inference has been limited to semi-honest security of small networks over tiny datasets such as MNIST or CIFAR. Even on MNIST/CIFAR, CRYPTFLOW outperforms prior work.

I. INTRODUCTION

Secure multiparty computation (or MPC) allows a set of mutually distrusting parties to compute a publicly known function on their secret inputs without revealing their inputs to each other. This is done through the execution of a cryptographic protocol which guarantees that the protocol participants learn only the function output on their secret inputs and nothing else. MPC has made rapid strides - from being a theoretical concept three decades ago [82], [35], to now being on the threshold of having real world impact. One of the most compelling use cases for MPC is that of machine learning (ML) - e.g. being able to execute inference over ML algorithms securely when the model and the query are required to be hidden from the participants in the protocol. There has been a flurry of recent works aimed at running inference securely with MPC such as SecureML [62], MinioNN [55], ABY^3 [60], CHET [27], SecureNN [79], Gazelle [49], Delphi [58], and so on. Unfortunately, these techniques are not easy-to-use by ML developers and have only been demonstrated on small deep neural networks (DNNs) on tiny datasets such as MNIST or CIFAR. However, in order for MPC to be truly ubiquitous for secure inference tasks, it must be both effortless to use and capable of handling large ImageNet [31] scale DNNs.

In this work, we present CRYPTFLOW, a first of its kind system, that converts TensorFlow [6] inference code into MPC protocols at the push of a button. By converting code in standard TensorFlow, a ubiquitous ML framework that is used in production by various technology companies, to MPC protocols, we significantly lower the entry barrier for ML practitioners and programmers to use cryptographic MPC protocols in real world applications. We make the following four contributions:

- First, we provide a compiler, called *Athos*, from TensorFlow to a variety of secure computation protocols (both 2 and 3 party) while preserving accuracy. In the absence of Athos, all prior works require *manually* re-implementing ML models in an MPC friendly low-level language/library, and hence, their evaluations have been limited to small benchmarks where this task is feasible.
- Second, we provide a semi-honest secure 3-party computation protocol, *Porthos*, that outperforms all prior protocols for secure inference and enables us to execute, for the first time, the inference of ImageNet scale networks in *about 30 seconds*.
- Third, assuming a minimally secure hardware which guarantees the integrity of computations, we show a novel technique, *Aramis*, that compiles any semi-honest secure MPC protocol to a malicious secure MPC protocol. Aramis only relies on these integrity checks and assumes no confidentiality guarantees for data residing within the hardware. Aramis enables the first implementations of DNN inference secure against malicious adversaries.
- Fourth, we demonstrate the ease-of-use, efficiency and scalability of CRYPTFLOW by evaluating on
(a) RESNET50 [40], which won the ImageNet Large Scale Visual Recognition Challenge in 2015 [31];
(b) DENSENET121 [43], a convolutional neural network that won the best paper at CVPR 2017.
These networks have heavily influenced the ML community with thousands of citations each. To demonstrate that CRYPTFLOW is immediately useful in healthcare, we

* Equal contribution

DOI 10.1109/SP40000.2020.00092

also evaluate CRYPTFLOW on DNNs used for prediction of lung diseases and diabetic retinopathy.

Our toolchain and all of our benchmarks are publicly available[1].We now describe our results in more detail.

A. Results

CRYPTFLOW outperforms prior work on ease-of-use, scalability, and efficiency. It automatically compiles TensorFlow code to MPC protocols with *no loss in classification accuracy*. This makes CRYPTFLOW the first secure inference system to produce a Top 1 accuracy of 76.45% and Top 5 accuracy of 93.23% for predictions running securely on the ImageNet dataset. Furthermore, in the 3-party (3PC) setting, this can be done in about 30 seconds with semi-honest security and about 2 minutes with malicious security. Prior work in the area of secure inference has been limited to small networks over tiny datasets such as MNIST or CIFAR. Moreover, these implementations are limited to security against weaker semi-honest adversaries, that are assumed not to modify the code of the MPC protocol. In contrast, our largest network RESNET-200 has 200 layers, 65 million parameters, over 1000 ImageNet classes, and the user can choose between semi-honest and malicious security – the latter also protects against adversaries who can deviate from the MPC protocol specification arbitrarily. We have evaluated CRYPTFLOW on secure inference over DNNs that are at least an order of magnitude larger than the state-of-the-art [58], [27], [72], [79], [62], [49], [19], [55], [60], [15], [71], [12]. Even on MNIST/CIFAR, CRYPTFLOW has lower communication complexity and is more efficient than prior and concurrent works [79], [60], [72], [12]. Furthermore, CRYPTFLOW is the first system to implement[2] malicious security for secure DNN inference. We show that the overhead of Aramis over semi-honest protocols is small and varies between 25% and 3X depending on the size of the computation. Moreover, by very conservative estimates, Aramis based secure DNN inference is faster than state-of-the-art malicious secure MPC inference protocols [4] by at least an order of magnitude (and also the maliciously secure MPC protocols for general computation [80], [81], [50]). Hence, on inference tasks, prior MPC protocols are either much slower than Aramis or fail to provide security against malicious adversaries.

B. Components of CRYPTFLOW

We describe the three components of CRYPTFLOW next.

Athos (Section III). Athos is a compiler that compiles TensorFlow inference code to secure computation protocols. There are several challenges in doing so. For optimizations (Section III-D), the compiler needs the dimensions of all the tensors occurring in the dynamic Python code. The compiler is designed to be modular (Section III-C) and it provides facilities for plugging in various MPC protocols. To demonstrate this modularity, we have implemented the following backends: ABY-based 2-party computation (2PC), Porthos-based semi-honest secure 3-party computation (3PC), and Aramis-based malicious secure 3-party computation.

The transformations implemented in Athos are sensitive to the performance of MPC protocols. For performance reasons all efficient secure computation protocols perform computation over fixed-point arithmetic - i.e., arithmetic over integers or arithmetic with fixed precision. This is in contrast to TensorFlow where computations are over floating-point values. Athos automatically converts TensorFlow code over floating-point values into code that computes the same function over fixed-point values. This compilation is done while *matching* the inference accuracy of floating-point code. Prior works ([62], [55], [49], [60], [79], [58]) in the area of running ML securely have performed this task by hand with significant losses in accuracy over floating-point code. Although these fixed-point conversions are feasible to do manually for one or two small benchmarks, this task quickly becomes intractable for large benchmarks and needs to be repeated for every new benchmark. Athos automates this tedious and error prone task.

Porthos (Section IV). Porthos is an improved semi-honest 3-party secure computation protocol (tolerating one corruption) that builds upon SecureNN [79]. Porthos makes two crucial modifications to SecureNN. First, SecureNN reduces convolutions to matrix multiplications and invokes the Beaver triples [13] based matrix multiplication protocol. When performing a convolution with filter size $f \times f$ on a matrix of size $m \times m$, the communication is roughly $2q^2f^2 + 2f^2 + q^2$ elements in the ring $\mathbb{Z}_{2^{64}}$, where $q = m - f + 1$. Porthos computes these Beaver triples by appropriately reshaping $m \times m$ and $f \times f$ matrices. This reduces the communication to roughly $2m^2 + 2f^2 + q^2$ ring elements. Typically the filter size, f, is between 1 and 11 and the communication of Porthos can be up to two orders of magnitudes lower than SecureNN. Additionally, in SecureNN, the protocols for non-linear layers (such as Rectified Linear Units (ReLU) and MaxPool) require the third party to send secret shares to the first two parties. In Porthos, we cut this communication to half by eliminating the communication of one of these shares. This reduces the communication in the overall ReLU and MaxPool protocols by 25%. Thus, by reducing the communication in both linear convolution layers and non-linear layers, the communication in Porthos is several GBs lower than SecureNN (Table VIII).

Aramis (Section V). Obtaining maliciously secure MPC protocols through cryptography can often be challenging and expensive – typically some sort of "proof of honest computation" must be provided by the parties for every step of the protocol. We present a novel technique, called Aramis, that compiles MPC protocols secure against semi-honest

[1] https://github.com/mpc-msri/EzPC

[2] ABY^3 [60] provided a theoretical protocol to convert their semi-honest protocol into a malicious secure protocol on much smaller benchmarks than CRYPTFLOW, but did not provide an implementation or experimental validation.

adversaries into MPC protocols that are secure against malicious adversaries, by leveraging secure hardware. We only require the hardware to provide code attestation and a secure signing functionality (that we use to sign and verify the protocol messages). Aramis has two attractive features: (a) it works in a strong adversarial threat model; and (b) it serves as a general technique that can work on a variety of semi-honest secure MPC protocols. In more detail:

(a) The threat model of Aramis is significantly stronger than the prior work on MPC using secure hardware [74], [69], [38], [11], [23], [29], [51], [32], [77], [83], [44]. Specifically, in our threat model, not only is the host operating system outside the Trusted Computing Base, but it is also allowed to observe the entire state of the hardware (including user data). In contrast, for security of the protocol, the prior works require that the hardware hides the state from the host and even if data is decrypted and computed upon inside the hardware, it cannot be viewed by the host. In Section V, we describe the Aramis threat model in more detail, formalize the secure hardware as an ideal functionality, provide a formal description of the malicious secure MPC protocols, and formally prove their security. The ideal functionality can potentially be realized using various hardware platforms that provide code attestation and signing, e.g., STM32H7, MediaTek MT3620, CEC1702, ARMTrustZone, Intel's SGX, etc. We provide a proof-of-concept implementation of Aramis by using SGX as the underlying secure hardware.

(b) Aramis is general and can be applied to any semi-honest secure MPC protocol. To demonstrate this, we derive malicious secure MPC protocols from both semi-honest GMW (2 party protocol) [35] and Porthos (3 party protocol). Porthos compiled with Aramis gives the first experimentally vetted maliciously secure protocol for neural network inference with at most 3X overhead over semi-honest security. While these were the semi-honest protocols we applied Aramis to, one could potentially obtain performant maliciously secure variants of several other recent semi-honest secure inference protocols (e.g. [49], [58], [7]), and MPC protocols for other applications [52], [47].

C. Organization of the paper

We provide an end-to-end walkthrough of our system to illustrate the overall toolchain in Section II. In Section III, we describe our compiler Athos. Section IV describes our improved 3-party semi-honest secure protocol for neural networks. We describe Aramis that compiles any semi-honest secure protocol into a malicious secure protocol, in Section V. We present all our experimental results in Section VI, related works in Section VII and conclude in Section VIII.

II. Motivating Example

In this section, we describe the end-to-end working of CrypTFlow through an example of logistic regression. The high-level toolchain is shown in Figure 1. We describe how code compilation happens from TensorFlow to MPC protocols.

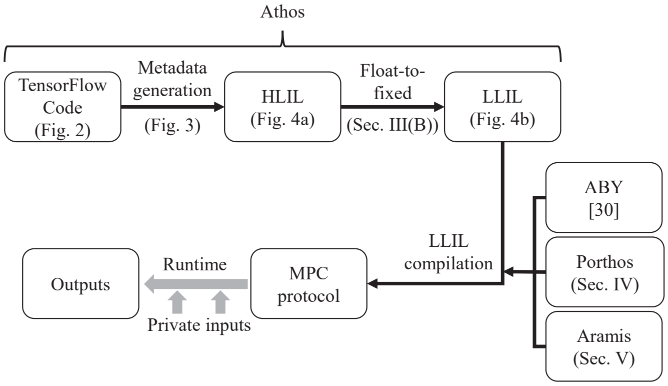


Fig. 1: CrypTFlow: End-to-end toolchain

```
# x is an MNIST image of shape (1,784).
# W and b are the model parameters.

print(tf.argmax(tf.matmul(x, W) + b, 1))
```

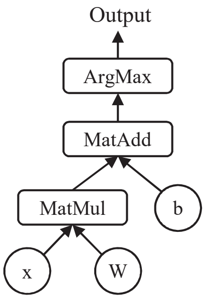
Fig. 2: Logistic Regression: TensorFlow snippet

The CrypTFlow toolchain takes as input code written in vanilla TensorFlow. For example, consider the code snippet for logistic regression over MNIST dataset in TensorFlow as shown in Figure 2. Our compiler first generates the TensorFlow graph dump (as shown in Figure 3a) as well as metadata to help compute the dimensions of all the tensors (Figure 3b). III-A provides more details on the frontend. Next, the TensorFlow graph dump is compiled into a high-level intermediate language HLIL. The code snippet for logistic regression in HLIL is shown in Figure 4a. Next, Athos' float-to-fixed converter translates the floating-point HLIL code to fixed-point code in a low-level intermediate language LLIL. This step requires Athos to compute the right precision to be used for maximum accuracy (Section III-B). Figure 4b shows the LLIL code snippet for logistic regression. The function calls in this sequence can be implemented with a variety of secure computation backends - e.g. ABY [30] for the case of 2-party secure computation, Porthos for the case of semi-honest 3-party secure computation (Section IV) and Aramis (Section V) for the malicious secure variant. Different backends provide different security guarantees and hence vary in their performance. For this example, the three backends take 227ms, 6.5ms, and 10.2ms respectively.

III. Athos

Athos compiles ML inference code written in TensorFlow to MPC protocols. It has the following main components:

- *Frontend.* Athos frontend compiles TensorFlow code to a high-level intermediate language (HLIL). HLIL supports floating-point tensors and sequence of function calls (corresponding to the TensorFlow nodes) that manipulate tensors. The main challenge in the frontend is to



(a)

Node	Outgoing dimensions
x	1 × 784
W	784 × 10
MatMul	1 × 10
b	1 × 10
MatAdd	1 × 10
ArgMax	1 × 1

(b)

Fig. 3: Logistic Regression: (a) TensorFlow graph definition (b) Metadata consisting of graph nodes and their outgoing dimensions

```
xW = MatMul(x, W);
xWb = MatAdd(xW, b);
output(ArgMax(xWb));
```

(a)

```
//Assume Athos chooses
//15 bit precision

xW = MatMul(x, W);
ScaleDown(xW, 15);
xWb = MatAdd(xW, b);
output(ArgMax(xWb));
```

(b)

Fig. 4: Logistic Regression in (a) floating-point: HLIL syntax (b) fixed-point: LLIL syntax

reconcile dynamic typing in TensorFlow to static typing in HLIL. TensorFlow code, written in Python, does not have tensor dimensions, whereas our HLIL has explicit tensor dimensions as it enables the compiler to perform analyses and optimizations.

- *Float-to-fixed converter.* While ML models use floating-point arithmetic, MPC protocols operate on fixed-point arithmetic. Rather than requiring the programmers to manually convert (or re-train) their models to integers, Athos performs the conversion automatically, without compromising on the inference accuracy.
- *Modular LLIL.* Athos compiles floating-point HLIL code to fixed-point code in a low-level intermediate language (LLIL). LLIL is a C-like imperative language that supports integer tensors, loops, conditionals, and functions. LLIL also makes it easier for different cryptographic backends to be plugged into Athos. It precisely specifies the interface that it requires the cryptographic protocols to implement, while providing a library for other operations. The LLIL is compiled down to the MPC protocol code.
- *Optimizations.* Athos implements MPC specific optimizations as well as several standard dataflow analyses and compiler optimizations. The design of HLIL and LLIL, and the choice of them being statically typed, is partly motivated by the requirements of these analyses.

Below we explain each of these components in detail.

A. Frontend and HLIL

Athos frontend compiles the input TensorFlow models to HLIL (described next) with explicit tensor dimensions. To obtain these dimensions, the frontend first runs TensorFlow code on one dummy input and generates TensorFlow metadata that has all the required information. The metadata is then translated to HLIL.

We discuss some details of the frontend. A plain dump of the TensorFlow metadata contains some nodes that are semantically irrelevant for actual inference, e.g. `identity`, `assign`, etc. To avoid representing these nodes in HLIL, we first prune the TensorFlow graph to remove such nodes, specifically we use the TensorFlow graph transform tool [1] for this purpose. Next, Athos desugars the remaining (tens of) TensorFlow nodes to HLIL, while keeping the number of functions in HLIL as small as possible. TensorFlow also supports "broadcasting" [76] that allows operations on tensors of incompatible dimensions and sizes. For example, due to broadcasting, addition of a four-dimensional tensor with a one-dimensional tensor is a valid operation. Athos frontend passes the broadcasting information to HLIL, which then accounts for it by compiling it to the appropriate LLIL library function call.

$$\begin{array}{rccl}
\text{Constant} & n & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
\text{Float constant} & r & ::= & n.n \\
\text{Type} & \hat{\tau} & ::= & \texttt{float} \mid \texttt{int} \mid \hat{\tau}[n] \\
\text{Matrix} & \hat{M} & ::= & \overline{r} \mid \overline{\hat{M}} \\
\text{Expression} & \hat{e} & ::= & n \mid x \mid \hat{M} \mid \hat{e_1} \oplus \hat{e_2} \mid x[\hat{e}] \\
\text{Program} & \hat{p} & ::= & \texttt{void main}\ ()\ \{\overline{\hat{\tau}\ x}\ ;\ \overline{f(\overline{\hat{e}})}\}
\end{array}$$

Fig. 5: HLIL syntax

Figure 5 shows the HLIL (we use $\overline{r}$ to denote sequences of floating-point constants, and similarly for other syntactic categories). It is a simple language of floating-point tensors ($\hat{M}$), with dimensions (n) and sizes as explicit type annotations ($\hat{\tau}[n]$), and the `main` is a sequence of variable declarations and function calls.

We next discuss how Athos performs float-to-fixed conversion on HLIL programs.

B. Float-to-fixed

As observed earlier, most ML models are expressed using floating-point, while MPC protocols operate on integers. For large models, we cannot expect the programmers to manually translate or re-train floating-point ML models to integer code (the common approach in literature on secure inference [62], [55], [49], [60], [79], [58], [72]). Furthermore, it is well-known that floating-point operations are much more inefficient than fixed-point when evaluated securely ([62], [60]) – we re-confirm this by performing two-party secure multiplication [28] using both fixed-point and floating-point arithmetic to showcase the difference. This is illustrated in Table I which shows the huge overheads associated with floating-point arithmetic. In future, if efficient protocols for floating-point become available then we can directly compile HLIL to them, but until then Athos automatically performs the translation.

The translation is parametrized by a scale parameter s that determines the precision. We discuss how this scale is set later in the section. Given a scale $s \in \mathbb{Z}$, we define a map $\rho_s : \mathbb{R} \rightarrow \mathbb{Z}_{2^b}$ that maps Reals to b-bit integers: $\rho_s(r) = \lfloor r \cdot 2^s \rfloor$. We abuse notation and also apply ρ_s to matrices M of Reals where the result is a point-wise application of ρ_s to each matrix element. In the output fixed-point code, every Real number r is represented by a b-bit integer. The Real representation of an integer n is given by $\frac{n}{2^s}$. The float-to-fixed conversion (for select cases) is described in the following algorithm (ScaleDown is described in Table II):

$$\begin{array}{rcl} F(\texttt{MatAdd}(A,B,C)) & = & \texttt{MatAdd}(A,B,C) \\ F(\texttt{MatMul}(A,B,C)) & = & \texttt{MatMul}(A,B,C); \\ & & \texttt{ScaleDown}(C,s) \end{array}$$

As an example of the conversion process, consider the program $M_1 * M_2$ that multiplies the row vector $M_1 = [400.1, 200.1]$ with the column vector $M_2 = [0.3, 0.1]^T$. Then in infinite precision Real arithmetic the result of the computation $400.1 * 0.3 + 200.1 * 0.1$ is 140.04. Single-precision floating-point arithmetic with 32 bits only has a 23-bit mantissa and computes the approximately correct result 140.040009. We use $0.1f$ to denote the floating-point number closest to the Real number 0.1. Given $s = 24$, $F(M_1 * M_2)$ results into the following program over integers

$$(\rho_{24}(400.1f) * \rho_{24}(0.3f) + \rho_{24}(200.1f) * \rho_{24}(0.1f)) >> 24$$

which results in the following computation with 64-bit integers

$$(6712564224 * 3357121024 + 5033165 * 1677721) >> 24$$

The final result is 2349481329 that represents the real number $\frac{2349481329}{2^{24}} = 140.04000002145767211914062$5 which is good approximation of the desired result 140.04. Although it is feasible to constuct examples where fixed-point computations can be imprecise, ML usually operates on normalized values and we have observed that Athos does not lose accuracy in practice (Table VI).

Athos, assigns the same bit-width b and the same scale s to all network parameters. While we could use different b and s, our experimental results show that same values for all parameters works quite well in practice. We keep the scale public for efficiency: division with 2^s when s is secret is much more expensive than when s is public. Moreover, scaling down operations (division by 2^s) cause loss of precision, as they lose significant bits, and hence need to be minimized. Therefore, Athos scales down only once per matrix multiplication and does not scale down matrix additions.

While we use machine integer width (64) for b, finding a good value of s is difficult. We explain the various tradeoffs that govern the choice of s and then discuss our solution.

Suppose, in our example, s is set too low: $s = 2$. Then $F([400.1f, 200.1f] * [0.3f, 0.1f])$ is $(1600*1 + 800*0) >> 2$, which represents the Real number $400/4 = 100$. This result is far from 140.04. Here, low scale values have lead to loss of significant bits. In particular, 0.1 has been rounded to zero causing an imprecise result. Ideally we want to set the scale to a large value so that the integers have many significant digits.

# Sequential Multiplications	Fixed (ms)	Float (ms)	Overhead
1	2.57	72.35	28.11x
10	4.88	278.8	57.1x
100	21.65	2735	126.34x
1000	199.6	25281.42	126.6x

TABLE I: Floating-point vs Fixed-point multiplication.

Next, suppose s is set to a very high value, e.g., 60. Then, the computation $\rho_{60}(400.1f) * \rho_{60}(0.3f)$ overflows 64-bit integers and the result is garbage (multiplication of these two large positive numbers would become a negative number).

Thus, scale can neither be very low nor very high; we need to find a sweet spot. To determine an appropriate value of s, we sweep over all its possible values $\{0, 1, \ldots, b-1\}$ and choose the value that leads to the best accuracy. For the example $400.1f * 0.3f + 200.1f * 0.1f$, the most accurate result is obtained at $s = 24$. In general, machine learning algorithms have a validation dataset that is used for hyperparameter tuning. We consider scale as a hyperparameter and select the scale that leads to a fixed-point classifier implementation that performs the best on the validation set. The scale chosen by Athos is a *leakage* function that depends on the weights of the model. Athos gives a methodical way of picking this scale that prior works did manually. Hence, leakage by Athos is similar to all prior works on secure inference.

C. Modular LLIL

$$\begin{array}{rccl} \text{Constant} & n & ::= & 0 \mid 1 \mid 2 \mid \ldots \\ \text{Type} & \tau & ::= & \texttt{int} \mid \tau[n] \\ \text{Matrix} & M & ::= & \overline{n} \mid \overline{M} \\ \text{Expression} & e & ::= & n \mid x \mid M \mid e_1 \oplus e_2 \mid x[e] \\ \text{Statement} & s & ::= & \tau\, x \mid x = e \mid \texttt{for}(x = e_1; x < e_2; x++)\{s\} \\ & & \mid & x[e_1] = e_2 \mid \texttt{if}(e, s_1, s_2) \mid s_1; s_2 \mid \texttt{return}\ e \\ & & \mid & f\ (\overline{e}) \mid d\ (\overline{e}) \\ \text{Global} & g & ::= & \texttt{extern}\ \tau\ d\ (\overline{\tau\, x}) \mid \tau\ f\ (\overline{\tau\, x})\{s\} \\ \text{Program} & p & ::= & \overline{g};\ \texttt{void main}\ ()\ \{s\} \end{array}$$

Fig. 6: LLIL syntax

Athos compiles HLIL to LLIL, a crypto-aware, C-like intermediate language that has only integer-valued tensors. Figure 6 shows the syntax of LLIL. This language has sufficient expressiveness required to implement ML inference tasks. In particular it supports arrays, basic arithmetic, loops, branching, functions, and extern declarations. LLIL makes the Athos interface to the MPC cryptographic protocols explicit. We observe that the tensor operations in a typical TensorFlow code fall into two categories: those that do not change the values but just copy the data around (e.g. squeeze to remove dimensions of size 1 from a tensor, pad to pad a tensor with various kinds of paddings, transpose to take the transpose of a tensor, and concat to concatenate two tensors into a single tensor), and those that compute new values.

For functions that do not manipulate shares (denoted by f), LLIL provides a library with their implementations that is automatically added as a prelude to LLIL programs. Changing the underlying crypto protocol does not require changes to these library functions and this library can be used by all crypto developers. These functions are implemented in LLIL and are compiled to C++ code.

Share-manipulating functions (`extern` d) are required to be implemented in the cryptographic backend. All a crypto developer needs to do is to implement these functions, and then she would be able to directly evaluate the protocols on ML models used in practice. We describe these functions with their signatures and intended semantics in Table II. Concretely, we provide three implementations of these functions: using the 2PC protocols of ABY [30], 3PC protocols of SecureNN [79], and Porthos (Section IV).

Finally, Athos compiles LLIL programs to C++ and links them with the cryptographic MPC protocol implementation.

D. Optimizations

Athos intermediate languages are designed to be amenable to static analysis. In particular, we have implemented several standard dataflow analyses and compiler optimizations [8]: reaching definitions, liveness analysis, and so on. These analyses help with optimizing memory utilization and we have observed savings reaching up to 80%. To demonstrate the ease of implementing analyses and optimizations, we provide an example each: (a) a peephole optimization ReLU MaxPool Switching on HLIL to improve efficiency of DNNs that use ReLU and MaxPool, and (b) an analysis Counting Scale Down operations on LLIL to determine the number of scale down operations done in order to prevent loss in accuracy (a similar analysis was done manually in [62], [79], [60]).

1) ReLU MaxPool Switching: Most TensorFlow developers have adopted the convention of expressing DNN layers using the MaxPool(ReLU(·)) idiom. For protocols like Porthos and SecureNN [79] that reduce ReLU and MaxPool to secure comparison protocols, ReLU(MaxPool(·)) can be much more efficient than MaxPool(ReLU(·)) as this significantly reduces the number of comparisons. As opposed to SecureNN, where this was done manually, we have built a peephole optimization pass on HLIL that replaces occurrences of $\texttt{MaxPool}(a, b, \texttt{ReLU}(A));$ with $\texttt{ReLU}(\texttt{MaxPool}(a, b, A));$. For example, if the input matrix A has dimensions $112 \times 112 \times 64$ and we compute a MaxPool with 2×2 windows. Then, the output matrix has dimensions $56 \times 56 \times 64$. Hence, the latter needs to compute only one fourth the number of ReLUs compared to the former. In this case, the optimized code is over 3× better in communication and over 2× faster in our experimental setup (Section VI).

2) Counting Scale Down operations: We describe an analysis to count the number of scale down operations in an LLIL code. The analysis uses an environment ρ that maps tensors to the number of elements they contain. This environment is populated using variable declarations in the code. The analysis makes a single pass over `main` and for each call $\texttt{ScaleDown}(\texttt{A}, \texttt{s})$ accumulates $\rho(A)$ into a counter. The final value of the counter provides the number of scale down operations in the code.

Note that this analysis is easy to describe as the LLIL code contains dimensions of all the tensors explicitly. Hence, the compiler can statically populate ρ. This analysis is impossible to perform on the TensorFlow Python code as the sizes of tensors are unknown at compile time.

IV. Porthos

We now describe Porthos, our improved secure 3PC protocol that provides semi-honest security against one corrupted party and privacy against one malicious corruption. The notion of privacy against malicious corruption (introduced by Araki *et al.* [10]) informally guarantees that privacy of inputs hold even against malicious party as long as none of the parties participating in the protocol learn the output of the computation (this is relevant, for example, when computation is offloaded to servers). Porthos builds upon SecureNN [79] but makes crucial modifications to reduce communication. We first describe our protocols that reduce communication and summarize concrete improvements in Table III.

We reduce communication for both linear as well as non-linear layers of DNNs. Linear layers include fully connected layers as well as convolutional layers. We improve the communication for convolutional layers and our optimization gains get better with larger filter sizes. With regards to non-linear layers (ReLU and MaxPool), we modify how two of the protocols in SecureNN are used – ComputeMSB and ShareConvert. As we explain below, this directly translates to better communication for both ReLU and MaxPool computations. At a very high level, we trade communication with compute by modifying the way certain shares are generated in the protocol.

Convolution. In SecureNN, secure computation of convolutional layers is done by reducing them to a (larger) matrix multiplication. As an example, 2-dimensional convolution of a 3×3 input matrix X (with single input channel and stride 1) with a filter Y of size 2×2 reduces to a matrix multiplication as follows:

$$\mathsf{Conv2d}\left(\begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \end{bmatrix}, \begin{bmatrix} y_1 & y_2 \\ y_3 & y_4 \end{bmatrix}\right) =$$

$$\begin{bmatrix} x_1 & x_2 & x_4 & x_5 \\ x_2 & x_3 & x_5 & x_6 \\ x_4 & x_5 & x_7 & x_8 \\ x_5 & x_6 & x_8 & x_9 \end{bmatrix} \times \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

In the above matrix multiplication, we call the left matrix (derived from X) as the “reshaped input” (say, X') and the right matrix (derived from Y) as the “reshaped filter” (say, Y'). The matrix multiplication is computed securely using a matrix Beaver triple [13], [62] based protocol. Later, the output can be reshaped to get the output of convolution in correct shape.

MatMul(int[L][M] A, int[M][N] B, int[L][N] C)	Multiply two tensors A and B and store results in C
MatAdd(int[L][M] A, int[L][M] B, int[L][M] C)	Add two tensors A and B into C
Conv(int[H][W][CI] A, int[FH][FW][CI][CO] F)	Convolve a tensor A with filter F
Avg/Max Pool(a, b, int[H][W][C] A)	Apply a stencil that computes the average/max value in windows of size $a \times b$ of tensor A.
ArgMax(int[M] A)	Compute the index with maximum value in A
FusedBatchNorm(int[K][L][M][N] A, int[N] B, int[N] C)	Returns $\forall k, l, m, n.B[n] \times A[k][l][m][n] + C[n]$
ReLU(int[M][N] A)	Returns $\forall i, j.\mathsf{Max}(A[i][j], 0)$
ScaleDown(int[M][N] A, k)	Arithmetic right shift each entry of A with k.

TABLE II: Share manipulating functions. These have been simplified for exposition by suppressing parameters such as padding and strides. For comprehensive signatures, see https://www.tensorflow.org/api_docs/python/tf/.

In this protocol, matrices being multiplied are masked by random matrices of same size and communicated and hence, the communication grows with the size of the matrices. We observe that this is quite wasteful for convolution because the reshaped input image (the first matrix in multiplication) has many duplicated entries (e.g., x_2 in row 1 and row 2) that get masked by independent random values. Let size of X be $m \times m$ and size of Y be $f \times f$. Then, the size of X' is $q^2 \times f^2$, where $q = m - f + 1$. In Porthos, we optimize the size of matrix-based Beaver triples for convolution by exploiting the structure of re-use of elements as the filter moves across the image. At a high level, we pick random matrix of size matching X for masking and communication only grows with size of X (i.e., m^2) instead of X' (i.e., $q^2 f^2$) in SecureNN.

Before, we describe our optimized protocol, we set up some notation. Let $\langle x \rangle_0^t$ and $\langle x \rangle_1^t$ denote the two shares of a 2-out-of-2 additive secret sharing of x over $\mathbb{Z}_t$ – in more detail, pick $r \xleftarrow{\$} \mathbb{Z}_t$, set $\langle x \rangle_0^t = r$ and $\langle x \rangle_1^t = x - r \pmod{t}$. $\langle x \rangle^t$ denotes a sharing of x over $\mathbb{Z}_t$. Reconstruction of a value x from its shares x_0 and x_1 is simply $x_0 + x_1$ over $\mathbb{Z}_t$. This generalizes to larger dimensions - e.g. for the $m \times n$ matrix X, $\langle X \rangle_0^t$ and $\langle X \rangle_1^t$ denote the matrices that are created by secret sharing the elements of X component-wise (other matrix notation such as $\mathsf{Reconst}^t(X_0, X_1)$ are similarly defined).

Let $\mathsf{Conv2d}_{m,f}$ denote a convolutional layer with input $m \times m$, 1 input channel, a filter of size $f \times f$, and 1 output channel. Our protocol for $\mathsf{Conv2d}_{m,f}$ is described in Algorithm 1, where $L = 2^\ell$, $\ell = 64$. Algorithms $\mathsf{ReshapeInput}$, $\mathsf{ReshapeFilter}$ and $\mathsf{ReshapeOutput}$ are used to reshape input, filter and output as described above and are formally described in Appendix A. Parties P_0 and P_1 start with shares of input matrix X and filter Y over $\mathbb{Z}_L$ That is, P_j holds $(\langle X \rangle_j^L, \langle Y \rangle_j^L)$ for $j \in \{0, 1\}$. In SecureNN, P_0 first reshapes $\langle X \rangle_0^L$ into $\langle X' \rangle_0^L$ by running $\mathsf{ReshapeInput}$. Then, it picks a random matrix $\langle A' \rangle_0^L$ of same size as X' and sends $\langle E' \rangle_0^L = \langle X' \rangle_0^L - \langle A' \rangle_0^L$ to P_1 that requires communicating $q^2 f^2$ elements. In Porthos, we optimize this as follows: P_0 picks a random matrix $\langle A \rangle_0^L$ of same size as X (Step 1) and sends $\langle E \rangle_0^L = \langle X \rangle_0^L - \langle A \rangle_0^L$ to P_1 (Step 4) that requires communicating m^2 elements only. Later, parties can reshape E locally to get E'. We reduce the communication by P_1 in a symmetric manner. Concretely, we reduce communication from $(2q^2f^2 + 2f^2 + q^2)\ell$ in SecureNN to $(2m^2 + 2f^2 + q^2)\ell$. This algorithm can be easily generalized to the setting where there are i input filters, o output filters, and different stride and padding parameters.

Algorithm 1 3PC protocol for $\mathsf{Conv2d}_{m,f}$

```
Input: $P_0$ holds $(\langle X \rangle_0^L, \langle Y \rangle_0^L)$ and $P_1$ holds $(\langle X \rangle_1^L, \langle Y \rangle_1^L)$, where $X \in \mathbb{Z}_L^{m \times m}$, $Y \in \mathbb{Z}_L^{f \times f}$.
Output: $P_0$ gets $\langle \mathsf{Conv2d}_{m,f}(X, Y) \rangle_0^L$ and $P_1$ gets $\langle \mathsf{Conv2d}_{m,f}(X, Y) \rangle_1^L$.
Common Randomness: $P_0$ & $P_1$ hold shares of a zero matrix $U$ of dimension $q \times q$, $q = m - f + 1$ . $P_0$ & $P_2$ hold a common PRF key $k_0$, and $P_1$ & $P_2$ hold a common PRF key $k_1$.
1) $P_0$ & $P_2$ use PRF key $k_0$ to generate random matrices $\langle A \rangle_0^L \in \mathbb{Z}_L^{m \times m}$, $\langle B \rangle_0^L \in \mathbb{Z}_L^{f \times f}$ and $\langle C \rangle_0^L \in \mathbb{Z}_L^{q \times q}$.
2) $P_1$ & $P_2$ use PRF key $k_1$ to generate random matrices $\langle A \rangle_1^L \in \mathbb{Z}_L^{m \times m}$ and $\langle B \rangle_1^L \in \mathbb{Z}_L^{f \times f}$.
3) $P_2$ computes $A = \langle A \rangle_0^L + \langle A \rangle_1^L$ and $B = \langle B \rangle_0^L + \langle B \rangle_1^L$. Let $A' = \mathsf{ReshapeInput}(A)$ and $B' = \mathsf{ReshapeFilter}(B)$. $P_2$ computes $\langle C \rangle_1^L = A' \cdot B' - \langle C \rangle_0^L$ and sends it to $P_1$.
4) For $j \in \{0, 1\}$, $P_j$ computes $\langle E \rangle_j^L = \langle X \rangle_j^L - \langle A \rangle_j^L$ and $\langle F \rangle_j^L = \langle Y \rangle_j^L - \langle B \rangle_j^L$ and sends to $P_{j \oplus 1}$.
5) $P_0$ & $P_1$ reconstruct $E$ and $F$ using exchanged shares.
6) For $j \in \{0, 1\}$, $P_j$ computes $\langle X' \rangle_j^L = \mathsf{ReshapeInput}(\langle X \rangle_j^L)$, $E' = \mathsf{ReshapeInput}(E)$, $\langle Y' \rangle_j^L = \mathsf{ReshapeFilter}(\langle Y \rangle_j^L)$, $F' = \mathsf{ReshapeFilter}(F)$.
7) For $j \in \{0, 1\}$, $P_j$ computes $\langle Z' \rangle_j^L = -jE' \cdot F' + \langle X' \rangle_j^L \cdot F' + E' \cdot \langle Y' \rangle_j^L + \langle C \rangle_j^L + \langle U \rangle_j^L$.
8) For $j \in \{0, 1\}$, $P_j$ outputs $\langle Z \rangle_j^L = \mathsf{ReshapeOutput}(\langle Z' \rangle_j^L)$.
```

Activation Functions. In SecureNN protocols for computing activations such as ReLU and MaxPool start with parties P_0 and P_1 having shares of values over $L = 2^{64}$. For both of these, parties run a protocol called $\mathsf{ComputeMSB}$ to evaluate most significant bit (MSB) of secret values. This protocol require shares over $L - 1$. So parties run a protocol called $\mathsf{ShareConvert}$ to convert shares over L to shares over $L - 1$. Both protocols $\mathsf{ComputeMSB}$ and $\mathsf{ShareConvert}$ require P_2 to send fresh shares of a value to P_0 and P_1. In SecureNN, both of these shares were picked by P_2 and explicitly communicated to P_0 and P_1. As mentioned before, shares of a value x are r and $x - r$, where r is a appropriately picked uniformly random value. We observe that since one of the shares is truly random, it can be computed as the output of a shared PRF key between P_2 and one of the parties, say P_0. This cuts the communication of this step to *half*. Moreover, since many

Protocol	Communication (SecureNN)	Communication (Porthos)
$\mathsf{Conv2d}_{m,i,f,o}$	$(2q^2f^2i + 2f^2oi + q^2o)\ell$	$(2m^2i + 2f^2oi + q^2o)\ell$
$\mathsf{ShareConvert}$	$4\ell \log p + 6\ell$	$3\ell \log p + 5\ell$
$\mathsf{ComputeMSB}$	$4\ell \log p + 13\ell$	$3\ell \log p + 9\ell$
ReLU	$8\ell \log p + 24\ell$	$6\ell \log p + 19\ell$
$\mathsf{MaxPool}_n$	$(8\ell \log p + 29\ell)(n-1)$	$(6\ell \log p + 24\ell)(n-1)$

TABLE III: Communication complexity of protocols; $q = m - f + 1$ and $\log p = 8$.

activations are computed in parallel, we can carefully "load-balance" this optimization between P_0 and P_1 to reduce the communication to half on the critical path. We implement this load-balance optimization and observe that this reduces the overall communication of $\mathsf{ShareConvert}$, $\mathsf{ComputeMSB}$, ReLU and $\mathsf{MaxPool}$ by 25%.

The revised table with comparison of overall communication complexity of all protocols with improvements over SecureNN are provided in Table III. $\mathsf{Conv2d}_{m,i,f,o}$ denotes a convolutional layer with input $m \times m$, i input channels, a filter of size $f \times f$, and o output channels. $\mathsf{MaxPool}_n$ computes the maximum value out of a list of n elements. p denotes a prime value strictly larger than 65 (set to 67 in SecureNN), with 8 bits being used to represent elements in $\mathbb{Z}_p$ (hence $\log p = 8$ in the table).

V. ARAMIS

In this section, we describe Aramis, a general technique to convert any semi-honest secure MPC protocol into a secure MPC protocol tolerating malicious corruptions by relying on secure hardware. The threshold of corrupted parties tolerated by the semi-honest protocol is retained in the malicious secure protocol by our technique.

Threat Model. We consider a strong threat model where not only does the adversary control the operating system of the corrupted parties (i.e., the host operating system is outside the Trusted Computing Base) but also observes the entire state of their secure hardware. Aramis makes a very minimal trust assumption of *integrity* on hardware, namely that of code attestation (the outputs generated by the hardware are indeed from the code that it attested to). This implicitly requires the hardware to possess a trusted component that can produce signatures and this signature scheme cannot be forged by the adversary. However, the adversary can see the state (i.e., all the code and the user data) of the hardware belonging to the corrupted parties, i.e., we *do not* assume *confidentiality* of state. Prior works [74], [69], [38], [11], [23], [29], [51], [32], [77], [83], [44] that combine MPC and hardware (SGX) make stronger trust assumption on the hardware of both confidentiality and integrity, and hence, provide security only in a weaker threat model where the hardware hides the data residing in it from the adversary.

Overview. At a very high level, Aramis exploits the following (well-known) observation: in order for a semi-honest protocol to be made maliciously secure, one must ensure that all messages sent by every party P_i are computed honestly according to the specification of the semi-honest protocol consistent with P_i's input and the transcript so far. The next observation we make is that if party P_i possesses hardware whose code can be attested by party P_j (and vice-versa), then P_j can obtain guarantees on the correctness of protocol messages sent by P_i as long as these messages are computed and signed by P_i's hardware. Using these observations, we can convert a semi-honest secure protocol into one that is maliciously secure by having every protocol message of P_i be computed by the trusted hardware that P_i executes. We shall now describe our techniques in more detail. We first describe the ideal functionality that is assumed out of the hardware in Section V-A. We then describe our technique in Section V-B. Finally, we provide an implementation of Aramis using Intel SGX as the underlying secure hardware. We explain how Intel SGX can realize the ideal functionality in Section V-C and challenges in porting semi-honest MPC protocols to SGX in Section V-D.

A. The attestation ideal functionality $\mathcal{F}_{\text{attest}}$

Description. We formally define the ideal functionality for attested executions in Figure 7. The functionality is parameterized by a signing key pair $(\mathsf{vk}, \mathsf{sk})$. Let $\mathsf{Sign}_{\mathsf{sk}}(m)$ denote the signing algorithm on message m and $\mathsf{Verify}_{\mathsf{vk}}(m, \sigma)$ denote verification of signature σ on message m. At a high level, this functionality allows users to specify a function g to the ideal functionality once using the Commit command. The functionality returns a token $\mathcal{T}_g$ generated as $\mathsf{Sign}_{\mathsf{sk}}(H(g))$, where H is a collision resistant hash function. Note that this token is publicly verifiable given g and vk. Let $\mathsf{state}_{\mathsf{ctr}}$ be an internal state that the functionality maintains, indexed by ctr – this state can be maintained by signing it along with ctr and verifying the signature of the state on every input message. When the functionality $\mathcal{F}_{\text{attest}}$ is initialized, the initial state state_0 is empty (or, ϵ). Subsequent invocations of the functionality is done on input w_{ctr} using the $\mathsf{Compute}$ command. The function g is a deterministic mapping from $(\mathsf{ctr}, w_{\mathsf{ctr}}, r_{\mathsf{ctr}}, \mathsf{state}_{\mathsf{ctr}-1})$ to $(y_{\mathsf{ctr}}, \mathsf{state}_{\mathsf{ctr}})$, where r_{ctr} is the required randomness. The functionality picks randomness r_{ctr}, evaluates g and provide a signature on the function output y_{ctr} using the signing key sk. Furthermore, $(y_{\mathsf{ctr}}, \mathsf{state}_{\mathsf{ctr}})$ is always given to party P such that $\mathsf{state}_{\mathsf{ctr}}$ contains r_{ctr} in clear and this ensures that there is no information hidden from P and we only assume correct execution of g. That is, the ideal functionality can evaluate functions and provide signed outputs and these outputs could have anyway been computed by party P given knowledge of $g, w_{\mathsf{ctr}}, r_{\mathsf{ctr}}, \mathsf{ctr}, \mathsf{state}_{\mathsf{ctr}}$, which are all known to P. Thereby, we only assume that the functionality will sign the output of g on the appropriate input and not hide any data from P. This significantly weakens what is assumed from the trusted hardware.

B. Semi-honest security to malicious security

Our technique takes any semi-honest secure MPC protocol and converts it into a malicious secure MPC protocol in

Functionality $\mathcal{F}_{\text{attest}}^{(\mathsf{vk},\mathsf{sk})}$

$\mathcal{F}_{\text{attest}}$ interacts with a party P.

- On input message (Commit, g) from P,
  1) Record $(\mathsf{Commit}, \mathsf{state}_0)$, where $\mathsf{state}_0 = \epsilon$;
  2) Send $(\mathsf{state}_0, \mathcal{T}_g)$ to P, where $\mathcal{T}_g = \mathsf{Sign}_{\mathsf{sk}}(H(g))$.
  3) Ignore further (Commit, g) messages.
- On input message $(\mathsf{Compute}, w_{\mathsf{ctr}})$ from P, retrieve $\mathsf{state}_{\mathsf{ctr}-1}$, pick required randomness r_{ctr} and compute $g(\mathsf{ctr}, w_{\mathsf{ctr}}, r_{\mathsf{ctr}}, \mathsf{state}_{\mathsf{ctr}-1})$ to obtain $(y_{\mathsf{ctr}}, \mathsf{state}_{\mathsf{ctr}})$ such that $\mathsf{state}_{\mathsf{ctr}}$ contains $(y_{\mathsf{ctr}}, r_{\mathsf{ctr}})$. Send $(y_{\mathsf{ctr}}, \mathsf{ctr}, \mathsf{Sign}_{\mathsf{sk}}(y_{\mathsf{ctr}}||\mathsf{ctr}), \mathsf{state}_{\mathsf{ctr}})$ to P.

Fig. 7: *The Authentication functionality* $\mathcal{F}_{\text{attest}}^{(\mathsf{vk},\mathsf{sk})}$.

the $\mathcal{F}_{\text{attest}}^{(\mathsf{vk},\mathsf{sk})}$−hybrid model. The idea is to have messages sent by every party P_i to every other party P_j in the semi-honest protocol be computed by the corresponding $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ functionality interacting with P_i, where $(\mathsf{vk}_i, \mathsf{sk}_i)$ are keys used by the functionality. These messages can be verified by functionality $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_j,\mathsf{sk}_j)}$ interacting with P_j. We assume that every $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ knows the verification key vk_j used by functionalities of all other parties P_j in a reliable manner. Later, we show how to achieve this through the use of remote attestation in the context of Intel SGX. We now set notation and describe the next message function of any semi-honest secure MPC protocol and how we modify it for our use.

Next message function. Let $\pi(\cdot)$ be the next message function of any semi-honest secure MPC protocol. $\pi(\cdot)$ takes the following values as input - two party ids i and j, input x_i, a round number ctr, randomness $r_{i,j,\mathsf{ctr}}$ and $\mathsf{Transcript}_i$, which includes the transcript of all messages sent and received by the party P_i so far. Given these, $\pi(\cdot)$ outputs $y_{\mathsf{ctr}}^{i,j}$, which is the message that P_i must send to P_j in round ctr and also updates $\mathsf{Transcript}_i$ appropriately. Additionally, $\pi(\cdot)$ takes message $y_{\mathsf{ctr}}^{j,i}$ sent by P_j to P_i at round ctr and update $\mathsf{Transcript}_i$ with this message. We now describe how to modify $\pi(\cdot)$ to $\pi^*(\cdot)$ to incorporate checks to detect malicious behavior.

Modified next message function. $\pi^*(\cdot)$, is the modified function that builds upon $\pi(\cdot)$ and we describe it for P_i.

1) For $\mathsf{ctr} = 1$, Let x_i be the input of P_i in $\pi(\cdot)$. Then, (ctr, x_i) is stored as state_1 (also called as $\mathsf{Transcript}_i^1$) and sent to P_i.
2) When $\pi^*(\cdot)$ receives a message $M = (y_{\mathsf{ctr}}^{j,i}, \mathsf{ctr}, \sigma)$ from party P_j, it runs $\mathsf{Verify}_{\mathsf{vk}_j}((y_{\mathsf{ctr}}^{j,i}, \mathsf{ctr}), \sigma)$. If verification succeeds, it appends M to $\mathsf{Transcript}_i$. Else, P_i aborts.
3) $\pi^*(\cdot)$ on input $(\mathsf{ctr}, \mathsf{state}_{\mathsf{ctr}-1}, j)$ computes the next message from P_i to P_j as follows: It checks that $\mathsf{state}_{\mathsf{ctr}-1}$ contains a valid transcript of all messages computed so far. If it verifies, it picks randomness $r_{i,j,\mathsf{ctr}}$ and runs $\pi(\mathsf{ctr}, \mathsf{state}_{\mathsf{ctr}-1}, j, r_{i,j,\mathsf{ctr}})$ to compute next message $y_{\mathsf{ctr}}^{i,j}$ and updated state $\mathsf{state}_{\mathsf{ctr}}$ (containing $r_{i,j,\mathsf{ctr}}$). Else it outputs $\perp$. Note that $\mathsf{state}_{\mathsf{ctr}-1}$ already contains input x_i, the input of party P_i.

Malicious MPC in the $\mathcal{F}_{\text{attest}}$−hybrid model. The malicious MPC protocol works as follows: Each party P_i invokes $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ with the command Commit using function $\pi^*(\cdot)$ described above and sends the received token $\mathcal{T}_{\pi^*}^{(i)}$ to other parties P_j. It receives similar tokens $\mathcal{T}_{\pi^*}^{(j)}$ from party P_j and verifies it under vk_j. Party P_i aborts if any of these verifications fail. If all verifications succeed, it proceeds with running $\pi^*(\cdot)$ inside $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ as described formally in Figure 8.

Protocol $\mathsf{Prot}_{\mathsf{malicious}}(P_1, \cdots, P_n)$

Party P_i with input x_i interacts with $\{P_j\}_{j \neq i}$ and $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ and does the following:

- Invokes $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ on (Commit, π^*) to receive $(\mathsf{state}_0^{(i)}, \mathcal{T}_{\pi^*}^{(i)})$ and sends $\mathcal{T}_{\pi^*}^{(i)}$ to all parties P_j, $j \neq i$.
- Receives $\mathcal{T}_{\pi^*}^{(j)}$ from P_j and runs $\mathsf{Verify}_{\mathsf{vk}_j}(H(\pi^*), \mathcal{T}_{\pi^*}^{(j)})$ for all $j \in [n] \setminus i$. Aborts if one of these checks fail.
- Invokes $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ on $(\mathsf{Compute}, x_i)$ to get $\mathsf{Transcript}_i^1$ containing input x_i.
- When P_i receives a message $M = (y_{\mathsf{ctr}}^{j,i}, \mathsf{ctr}, \sigma)$ from party P_j, it invokes $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ on $(\mathsf{Compute}, (y_{\mathsf{ctr}}^{j,i}, \mathsf{ctr}, \sigma))$ and receives updated transcript or $\perp$ (and aborts).
- When P_i needs to send next message to P_j it invokes $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ on $(\mathsf{Compute}, j)$ and receives $(y_{\mathsf{ctr}}^{i,j}, \mathsf{ctr}, \sigma)$ along with updated transcript and randomness used. Here, σ is a signature on $(y_{\mathsf{ctr}}^{i,j}, \mathsf{ctr})$ under sk_i. It sends $(y_{\mathsf{ctr}}^{i,j}, \mathsf{ctr}, \sigma)$ to P_j.
- When P_i has no more messages to send in $\pi(\cdot)$, it computes the output of the function from $\mathsf{Transcript}_i$.

Fig. 8: *Malicious secure MPC* $\mathsf{Prot}_{\mathsf{malicious}}(P_1, \cdots, P_n)$.

Functionality $\mathcal{F}_{\text{mpc}}^f(P_1, \cdots, P_n)$

$\mathcal{F}_{\text{mpc}}^f$ interacts with parties $\{P_1, \cdots, P_n\}$ and the adversary $\mathcal{S}$.

- On input message x_i from P_i record x_i and ignore further x_i from P_i
- Upon receiving x_i from all $P_i, i \in [n]$, compute $y = f(x_1, \cdots, x_n)$ and send to $\mathcal{S}$.
- Upon receiving (i, Send) or $(i, \perp)$ from $\mathcal{S}$, send y or $\perp$ to P_i.

Fig. 9: *The MPC functionality* $\mathcal{F}_{\text{mpc}}^f$.

Malicious Security. Next, we prove that if π is secure against semi-honest adversaries, then the protocol described in Figure 8 is an MPC protocol secure against malicious adversaries with the same corruption threshold. We prove the following result using the standard simulation paradigm in Appendix E.

Theorem 1. *Let $\pi(\cdot)$ be a semi-honest secure MPC protocol securely realizing $\mathcal{F}_{mpc}^f$. Then, protocol* $\mathsf{Prot}_{\mathsf{malicious}}(P_1, \cdots, P_n)$ *described in Figure 8 securely realizes $\mathcal{F}_{mpc}^f$ in the $\mathcal{F}_{attest}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$−hybrid model (with $i \in [n]$) against malicious adversaries.*

C. Realizing $\mathcal{F}_{\text{attest}}$

We note that the ideal functionality assumed out of the hardware can potentially be realized using various hardware platforms that provide code attestation and secure signing, e.g., STM32H7, MediaTek MT3620, CEC1702, ARMTrustZone, Intel SGX, etc. In this work, we provide an implementation of Aramis based on Intel SGX.

SGX allows a host to create a protected region known as an enclave. Intel gives integrity guarantees, that is, the code and the data residing in the enclave, once attested, cannot be modified by the host or the operating system. When SGX receives a Commit command (Figure 7) for a function g, then it creates an enclave with code g. Randomness r_{ctr} of Figure 7 can be sampled in SGX using `sgx_read_rand` command. The attestation token $\mathcal{T}_g$ is generated by SGX communicating with Intel's Attestation Service (IAS) and this token is publicly verifiable given g and public verification key corresponding to Intel's Report Signing Key. The key-pair $(\mathsf{vk}, \mathsf{sk})$ for ECDSA signature scheme is also generated inside the enclave and the verification key vk is sent as payload to IAS during the generation of the attestation token. The token $\mathcal{T}_g$ contains the verification key vk in the clear and this vk can be used to verify the signed outputs y_{ctr}. Now, on receiving the $\mathsf{Compute}$ command, the enclave starts executing the code of g and produces outputs signed under sk.

While running MPC in the $\mathcal{F}_{\text{attest}}$-hybrid, we require the enclave to reliably have verification keys used by enclaves of all other parties. This can be done by attaching the following prelude to π^* (the code running inside SGX): Read the tokens of all parties, parse them to obtain the verification keys, and verify the signature on the tokens using verification key of Intel's Report Signing key. Note that since all the parties are running the same function π^* (appended with this prelude), they can compute the hash of π^* locally and compare it with the hash in the tokens (which has been signed by Intel's IAS) of all the other parties, proceeding only if they all match perfectly.

D. Implementation challenges with Intel SGX

We outline some of the key challenges in implementing MPC between multiple SGX enclaves that involve multiple rounds of interaction and operate over large volumes of data.

1) Memory constraints in SGX: In SGX, all the enclave content, including code, and related data is stored in a special region of memory known as the Enclave Page Cache (EPC). The size of EPC is fixed in BIOS and can have a maximum size of 128MB. Typically, paging facilitates the execution of enclaves which cannot fit in EPC and any page that is evicted out is encrypted before storing it on unprotected memory [45]. This additional overhead has detrimental effects on the overall performance of the enclave application. We reduce the working set of secure inference tasks to limit these overheads.

- *ReLU and MaxPool functions:* We split the computation of memory-intensive non-linear functions into chunks that fit in the EPC to avoid paging. However, lower chunk sizes increase the number of rounds, and so, the chunk sizes must be carefully selected. For ResNet50, we set the chunk sizes for ReLU and MaxPool layers to be 40 MB and 10 MB respectively. For our network configurations, the increase in rounds is justified by the elimination of paging costs and reduction in end-to-end runtimes.
- *Convolution and Matrix Multiplication functions:* For the linear functions, we block the matrices into smaller ones, process the blocks, and aggregate them. We ensure that individual blocks fit in EPC.
- *Liveness Analysis:* Athos implements liveness analysis (Section III-D) which reduces the memory footprint of the compiled DNNs. For example, the memory footprint of ResNet50 reduces from 1100 MB to 397 MB due to liveness analysis. When chunking and liveness analysis are done together, the memory footprint of ResNet50 comes down to 297MB.

2) Porting Interactive Protocols to SGX: To the best of our knowledge, we are the first work to implement highly interactive protocols in SGX and this comes with unique challenges. For example, whenever data is passed across the enclave's protected memory region, it has to be *marshalled* in/out of the region. The performance of marshalling depends on the size of the parameters crossing the bridge. Larger parameters imply slower marshalling [45], while smaller parameters increase the total numbers of cross-bridge calls (which have an overhead of their own). Thus, we tune the payload size carefully. We also implement the techniques in [78] for optimizing communication involving enclaves.

VI. Experiments

Overview. In this section, we present our experimental results. First, in Section VI-A, we use CrypTFlow to securely compute inference on the ImageNet dataset using the following TensorFlow programs: ResNet50[3] and DenseNet121[4]. We also show that the performance of semi-honest and malicious protocols generated by CrypTFlow scale linearly with the depth of DNNs. Second, in Section VI-B, we show that CrypTFlow outperforms prior works on secure inference of DNNs. Next, we evaluate each component of CrypTFlow in more detail. In Section VI-C, we show that the fixed-point code generated by Athos matches the accuracy of floating-point ResNet50 and DenseNet121. We show in Section VI-D how the optimizations in Porthos help it outperform prior works in terms of communication complexity and overall execution time. In Section VI-E, we show the overhead of obtaining malicious secure MPC (over semi-honest security) using Aramis for GMW [35] and Porthos. We show Aramis-based malicious secure inference outperforms pure crypto-based malicious secure protocols by huge margins in Section VI-E1. Finally, in section VI-F, we discuss two case-studies of running CrypTFlow on DNNs for healthcare. We begin by providing details of the systems

[3] https://github.com/tensorflow/models/tree/master/official/r1/resnet

[4] https://github.com/pudae/tensorflow-densenet

Benchmark	Semi-Honest (s)	Malicious (s)	Comm. (GB)
ResNet50	25.9	75.4	6.9
DenseNet121	36.0	112.9	10.5

TABLE IV: CrypTFlow: ImageNet scale benchmarks.

used to run our experiments.

System Details. All our large benchmark experiments are in a LAN setting on 3.7GHz machines, each with 4 cores and with 16 GB of RAM running Linux Ubuntu 16.04. The measured bandwidth between each of the machines was at most 377 MBps and the latency was sub-millisecond. Since we wanted to use the same machines to benchmark both our semi-honest as well as our malicious secure protocols, we were constrained to use machines that had Intel SGX enabled on them - this led to machines that had considerably lower bandwidth between them (377 MBps) than those normally used by prior works in the area (e.g. [60], [12] used networks with bandwidth of 1.5 GBps). For Aramis, we used Intel SGX SDK version 2.4. The compilation time of CrypTFlow is around 5 sec for ResNet50, 35 sec for DenseNet121 and 2 minutes for ResNet200.

A. Secure Inference on ImageNet

We briefly describe our benchmarks and then present performance results.

1) ResNet50 [40] is a network that follows the residual neural network architecture. The residual nodes employ "skip connections" or short cuts between layers. It consists of 53 convolution layers with filters of size up to 7×7, and 1 fully connected layer of size 2048×1001. The activation function between most layers is batch normalization (Appendix B) followed by ReLU. After the first convolutional layer, the activation function also includes a MaxPool.
2) DenseNet121 [43] is a form of residual neural network that employs several parallel skips. It consists of 121 convolutional layers with filters of size up to 7×7. The activation function between these layers is usually batch normalization, followed by ReLU. Some layers also use MaxPool or AvgPool.

Performance. Table IV shows performance of CrypTFlow on these benchmarks. We measure communication as total communication between all 3 parties - each party roughly communicates a third of this value. The communication in semi-honest secure and malicious secure inference is almost the same. Thus demonstrating that ImageNet scale inference can be performed in about 30 seconds with semi-honest security and in under two minutes with malicious security. The malicious protocol of CrypTFlow is about 3x slower than the semi-honest version.

Scalability. We show that the running time of CrypTFlow-based protocols increases linearly with the depth of DNNs.

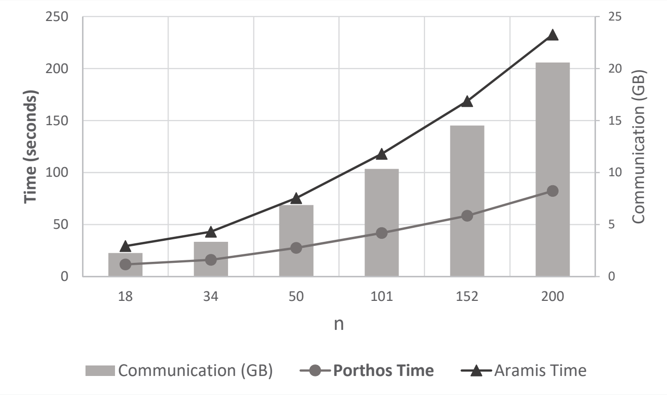


Fig. 10: Scalability of CrypTFlow on ResNet-n.

We compile ResNet-n (where n, the approximate number of convolutional layers, varies from 18 to 200) with CrypTFlow and evaluate with both semi-honest (Porthos) and malicious secure protocols (Aramis) in Figure 10. Our largest benchmark here is ResNet-200, the deepest version of ResNet on the ImageNet dataset [41], which has 65 million parameters. Other ResNet-n benchmarks have between 11 to 60 million parameters [5]. We observe that the communication and runtime increase linearly with depth. Even with increasing depth, the overhead of malicious security (over semi-honest security) remains constant at about 3X.

B. Comparison with prior work

In this section, we show that CrypTFlow outperforms prior works on secure inference of DNNs on the benchmarks they consider, i.e., tiny 2–4 layer DNNs over the MNIST and CIFAR-10 datasets. We stress that these benchmarks are very small compared to the ImageNet scale DNNs discussed above. In order to provide a fair comparison, for these experiments, we use a network with similar bandwidth as prior works (1.5 GBps) and machines with similar compute (2.7 GHz).

Table V shows that Porthos outperforms prior (ABY^3 [60], Chameleon[6] [72], and SecureNN [79]) and concurrent (QuantizedNN [12]) semi-honest secure 3PC works on the MNIST dataset. It is well-known that 3PC-based techniques like Porthos are much faster than techniques based on 2PC and FHE. We relegate comparison between Porthos and 2PC/FHE works to Appendix D. We omit comparisons with [22], [20] as their published MSB protocol was incorrect [21].

C. Athos experiments

Accuracy of Float-to-Fixed. We show that Athos generated fixed-point code matches the accuracy of floating-code on ResNet50 and DenseNet121 in Table VI. The table also shows the precision or the scale that is selected by Athos (Section III-B). We observe that different benchmarks require

[5] Specifically, 11, 22, 25, 44 and 60 million parameters for ResNet-n for n = 18, 34, 50, 101, and 152 respectively.

[6] Chameleon is a 2PC protocol in the online phase but requires a trusted third party in the offline phase. We report overall time here.

Benchmark	[60]	[12]	[79]	[72]	Porthos
Logistic Regression	4	—	—	-	2.7
SecureML (3 layer)	8	20	17	-	8
MiniONN (4 layer)	-	80	47	2240	34
LeNet (4 layer)	-	120	79	-	58

TABLE V: Comparison with 3PC on MNIST dataset with ABY^3 [60], QuantizedNN [12], SecureNN [79], and Chameleon [72]. All times in milliseconds.

Benchmark	Float Top 1	Fixed Top 1	Float Top 5	Fixed Top 5	Scale
ResNet50	76.47	76.45	93.21	93.23	12
DenseNet121	74.25	74.33	91.88	91.90	11

TABLE VI: Accuracy of fixed-point vs floating-point.

different precision to maximize the classification accuracy and that the technique of "sweeping" through various precision levels is effective. We show how accuracy varies with precision in Appendix C. Evaluating accuracy also helps validate the correctness of our compilation [63].

Modularity. Since CrypTFlow is modular, we can compile it to various MPC backends. To demonstrate this ability, we also add a 2PC semi-honest secure protocol ABY [30] to CrypTFlow. The performance with this backend is in Table VII. We ran logistic regression (LR) as well as a small LeNet network [53] which comprises of 2 convolutional layers (with maximum filter size of 5×5) and 2 fully connected layers, with ReLU and MaxPool as the activation functions. This evaluation shows that CrypTFlow can be easily used for a variety of backends.

D. Porthos experiments

Since Porthos builds on SecureNN, we compare them in mode detail. As described earlier, Porthos improves over the communication complexity of SecureNN [79] both for convolutional layers as well as for non-linear activation functions. We have already compared SecureNN and Porthos on benchmarks considered in SecureNN in Table V. Additionally, we also compare Porthos and SecureNN on ImageNet scale benchmarks in Table VIII. For this purpose, we add the code of SecureNN available at [5] as another backend to CrypTFlow. These results show that Porthos improves upon the communication of SecureNN by a factor of roughly 1.2X–1.5X and the runtime by a factor of roughly 1.4X–1.5X.

Benchmark	CrypTFlow (s)	Communication (MB)
LogisticRegression	0.227	25.5
LeNet Small	47.4	2939

TABLE VII: CrypTFlow compilation to 2PC on MNIST.

Benchmark	SecureNN (s)	Porthos (s)	SecureNN Comm. (GB)	Porthos Comm. (GB)
ResNet50	38.36	25.87	8.54	6.87
DenseNet121	53.99	36.00	13.53	10.54

TABLE VIII: Porthos vs SecureNN.

Benchmark	GMW (ms)	Aramis (ms)	Overhead
$\mathsf{IP}_{10,000}$	464	638	1.37x
$\mathsf{IP}_{100,000}$	2145	3318	1.54x
Add_{32}	279	351	1.25x
Mult_{32}	354	461	1.30x
$\mathsf{Millionaire}_{32}$	292	374	1.28x

TABLE IX: Semi-honest GMW vs Malicious Aramis.

E. Aramis experiments

We applied Aramis to both the 2-party GMW protocol [35] (using the codebase [2], based on [24]) as well as Porthos. The results for different functions using the GMW protocol are presented in Table IX. IP_n denotes the inner product of two n-element vectors over $\mathbb{F}_2$, Add_{32} and Mult_{32} denote addition and multiplication over 32 bits respectively, and $\mathsf{Millionaire}_{32}$ denotes the millionaires problem that compares two $32-$bit integers x and y and outputs a single bit denoting whether $x > y$. The overheads of Aramis-based malicious security, are within 54% of the semi-honest protocol. Table IV and Figure 10 evaluate Aramis with Porthos.

1) Comparison with crypto-only malicious MPC: We demonstrate that Aramis based malicious secure protocols are better suited for large scale inference tasks compared to pure cryptographic solutions. We compare the performance of Porthos compiled with Aramis and the concurrent work of QuantizedNN [12] that uses the MP-SPDZ [4] framework to also provide a malicious secure variant of their protocol. Both these approaches provide security for the same setting of 3PC with 1 corruption. On the four MNIST inference benchmarks A/B/C/D in the MP-SPDZ repository, Aramis is 10X/46X/44X/15X faster.

F. Real world impact

We discuss our experience with using CrypTFlow to compile and run DNNs used in healthcare. These DNNs are available as pre-trained Keras models. We converted them into TensorFlow using [3] and compiled the automatically generated TensorFlow code with CrypTFlow.

a) Chest X-Ray: In [85], the authors train a DenseNet121 to predict lung diseases from chest X-ray images. They use the publicly available NIH dataset of chest X-ray images and end up achieving an average AUROC score of 0.845 across 14 possible disease labels. During secure inference, we observed no loss in accuracy and the runtime is similar to the runtime of DenseNet121 for ImageNet.

b) Diabetic Retinopathy CNN: Diabetic Retinopathy (DR), one of the major causes of blindness, is a medical condition that leads to damage of retina due to diabetes [64]. In recent times, major tech companies have taken an interest in using DNNs for diagnosing DR from retinal images [64], [37]. Predicting whether a retina image has DR or not can be done securely in about 30 seconds with CrypTFlow.

VII. Related Work

High level languages. CrypTFlow is the first system to compile pre-defined TensorFlow code to secure MPC protocols. There have been prior works that compile from lower-level, domain-specific languages to MPC. Examples include Fairplay [56], Wysteria [70], ObliVM [54], CBMC-GC [42], SMCL [66], [59], Sharemind [16], EzPC [19], and SPDZ [9]. Reimplementing large DNNs in the input format of these tools is a formidable task. PySyft [73] and TF-Encrypted [26] are ongoing efforts that also aim to compile DNNs to MPC protocols. In contrast to CRYPTFLOW that compiles standard TensorFlow code, these works require reimplementing the DNNs in a dialect of PyTorch/TensorFlow. To the best of our knowledge, these systems have not been evaluated on ImageNet scale tasks.

Fixed-point in MPC. Although the use of fixed-point for secure computations is well-known [68], prior works on secure inference have addressed the float-to-fixed problem by either generating a fixed-point model by hand ([62], [55], [49], [60], [79], [58], [72]), or by using non-standard training algorithms that output fixed-point models ([71], [7]). Both of these approaches are unsatisfactory. In particular, some of the challenges that one would face with the latter include: a) the need to train again on the whole training data which is both computationally expensive, and impossible if the training data is unavailable; and b) training algorithms that generate integer models is still an active research area and an overwhelming majority of ML training algorithms still generate floating-point models. Athos alleviates all these problems by working with a trained model and being completely oblivious to the training procedure. The ML users can train their networks in the manner they see fit and then use Athos to get fixed-point code. Finally, even with retraining, TensorFlow-generated binary/integer networks suffer significant accuracy loses [48] whereas Athos matches the accuracy of floating-point models.

Float-to-fixed. The research literature in float-to-fixed for digital signal processors is rich and spans several decades. However, it is only recently that these schemes have been adapted to machine learning. Some recent float-to-fixed schemes [36], [65], [48] show promise by quantizing floating-point models to 8-bit or 16-bit integers. One could potentially use one of these systems in place of our float-to-fixed component – however, their compatibility with MPC protocols [60], [79] is unclear. Additionally, since we use higher bit-width of 64, not surprisingly, the accuracy of CRYPTFLOW is better.

Secure Machine Learning. There has been a flurry of recent results ([61], [75], [57], [25], [44]) in the area of secure machine learning, both in the 2-party [17], [67], [34], [55], [49], [58], [84], as well as in the 3-party setting [72], [60], [79], [12]. The most relevant to our work are ABY^3 [60] and SecureNN [79] that both provide 3-party semi-honest secure computation protocols for a variety of neural network inference and training algorithms, with somewhat similar performance guarantees. Porthos, our 3-party semi-honest protocol, outperforms both these works. We also remark that there have been other recent works [71], [75], [7], [33], [27], [15], [14], [39], that modify the inference or training algorithms in order to obtain performance benefits. These are applicable only to specialized benchmarks. For example, the works that use fully homomorphic encryption (e.g., [27], [15], [14]) do not support secure evaluation of ReLUs, XONN [71] requires DNNs to have binary weights, etc. On the other hand, we focus on standard inference algorithms and CRYPTFLOW has much wider applicability.

Hardware-based security. Our work is the first to provide experimentally validated malicious secure inference of ML algorithms at the scale of RESNET50. As discussed earlier, we achieve this by relying on minimally secure hardware to provide integrity. Prior works that use hardware enclaves for secure computation [74], [69], [38], [11], [23], [29], [51], [32], [77], [83], [44] assume that the enclave hides all data residing in it from the host. Thus, unlike Aramis, these systems are not secure against an adversary that can observe the SGX state. The only prior work that assumes a weaker trust assumption from the hardware is that of [78]. Similar to our work, they assume that the hardware provides integrity. However, their work is in the context of zero-knowledge proofs and other fundamentally asymmetric primitives that require only one enclave and not interactive protocols between multiple enclaves.

VIII. CONCLUSION

CRYPTFLOW is the first end-to-end system that translates high-level TensorFlow inference code to MPC protocols. It has 3 components - a) compiler from TensorFlow to MPC, b) an improved semi-honest 3PC protocol for DNNs, and c) a generic technique to convert semi-honest secure protocols to malicious secure ones. Using CRYPTFLOW, we demonstrate the first instance of secure inference on large benchmarks such as RESNET50 and DENSENET121 on the ImageNet dataset with both semi-honest (in about thirty seconds) and malicious security (in less than two minutes). CRYPTFLOW's modular design supports a variety of backends, and we hope that it can serve as a testbed for benchmarking new MPC protocols in the area.

Going forward, we would like to plugin protocols like SPDZ [4] and Delphi [58] in CRYPTFLOW. Our more ambitious goal is to extend CRYPTFLOW to support TensorFlow training. It is a challenging problem since in the absence of the GPU support, the overheads of MPC protocols for secure training can be prohibitive.

IX. ACKNOWLEDGEMENTS

We thank our shepherd Xiao Wang, and anonymous reviewers for their valuable feedback. We also thank Sridhar Gopinath, Aayan Kumar, Wonyeol Lee, Sundararajan Renganathan, and Kapil Vaswani for helpful discussions.

REFERENCES

[1] "TensorFlow Graph Transform Tool," https://github.com/tensorflow/tensorflow/blob/master/tensorflow/tools/graph_transforms/README.md.

[2] "GMW codebase," 2012. [Online]. Available: http://www.ee.columbia.edu/~kwhwang/projects/gmw.html

[3] "Keras to TensorFlow," https://github.com/amir-abdi/keras_to_tensorflow, 2019.

[4] "Multi-Protocol SPDZ: Versatile framework for multi-party computation," 2019. [Online]. Available: https://github.com/data61/MP-SPDZ

[5] "SecureNN: 3-Party Secure Computation for Neural Network Training," https://github.com/snwagh/securenn-public, 2019.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: https://arxiv.org/abs/1603.04467

[7] N. Agrawal, A. S. Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: Two-Party Secure Neural Network Training and Prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, 2019, pp. 1231–1247.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[9] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida, "Generalizing the SPDZ Compiler For Other Protocols," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 880–895.

[10] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, "High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, Vienna, Austria, October 24-28, 2016*, 2016, pp. 805–817.

[11] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A. Sadeghi, G. Scerri, and B. Warinschi, "Secure Multiparty Computation from SGX," in *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, 2017, pp. 477–497.

[12] A. Barak, D. Escudero, A. Dalskov, and M. Keller, "Secure Evaluation of Quantized Neural Networks," Cryptology ePrint Archive, Report 2019/131, 2019, https://eprint.iacr.org/2019/131.

[13] D. Beaver, "Efficient Multiparty Protocols Using Circuit Randomization," in *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, 1991, pp. 420–432.

[14] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data," Cryptology ePrint Archive, Report 2019/947, 2019, https://eprint.iacr.org/2019/947.

[15] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*, 2019, pp. 3–13.

[16] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A Framework for Fast Privacy-Preserving Computations," in *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, 2008, pp. 192–206.

[17] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine Learning Classification over Encrypted Data," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[18] R. Canetti, "Security and Composition of Multiparty Cryptographic Protocols," *J. Cryptology*, vol. 13, no. 1, pp. 143–202, 2000.

[19] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, "EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning," in *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, 2019, pp. 496–511.

[20] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, 2019, pp. 81–92.

[21] ——, "ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction," Cryptology ePrint Archive, Report 2019/429, 2019, https://eprint.iacr.org/2019/429.

[22] H. Chaudhari, R. Rachuri, and A. Suresh, "Trident: Efficient 4PC Framework for Privacy Preserving Machine Learning," in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[23] J. I. Choi, D. J. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. B. Butler, and P. Traynor, "A Hybrid Approach to Secure Function Evaluation Using SGX," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, 2019, pp. 100–113.

[24] S. G. Choi, K. Hwang, J. Katz, T. Malkin, and D. Rubenstein, "Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces," in *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, 2012, pp. 416–432.

[25] H. Corrigan-Gibbs and D. Boneh, "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, 2017, pp. 259–282.

[26] M. Dahl, J. Mancuso, Y. Dupis, B. Decoste, M. Giraud, I. Livingstone, J. Patriquin, and G. Uhma, "Private Machine Learning in TensorFlow using Secure Computation," *CoRR*, vol. abs/1810.08130, 2018. [Online]. Available: http://arxiv.org/abs/1810.08130

[27] R. Dathathri, O. Saarikivi, H. Chen, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, 2019, pp. 142–156.

[28] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni, "Automated Synthesis of Optimized Circuits for Secure Computation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015, pp. 1504–1517.

[29] D. Demmler, T. Schneider, and M. Zohner, "Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, 2014, pp. 893–908.

[30] ——, "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[31] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*, 2009, pp. 248–255.

[32] S. Felsen, A. Kiss, T. Schneider, and C. Weinert, "Secure and Private Function Evaluation with Intel SGX," in *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, 2019, pp. 165–181.

[33] S. Garg, Z. Ghodsi, C. Hazay, Y. Ishai, A. Marcedone, and M. Venkitasubramaniam, "Outsourcing Private Machine Learning via Lightweight Secure Arithmetic Computation," *CoRR*, vol. abs/1812.01372, 2018. [Online]. Available: http://arxiv.org/abs/1812.01372

[34] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016, pp. 201–210.

[35] O. Goldreich, S. Micali, and A. Wigderson, "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority," in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, 1987, pp. 218–229.

[36] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling KB-Sized Machine Learning Models to Tiny IoT Devices," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, 2019, pp. 79–95.

[37] V. Gulshan, L. Peng, M. Coram, M. C. Stumpe, D. Wu, A. Narayanaswamy, S. Venugopalan, K. Widner, T. Madams, J. Cuadros, R. Kim, R. Raman, P. C. Nelson, J. L. Mega, and D. R. Webster, "Development and Validation of a Deep Learning Algorithm for Detection of Diabetic Retinopathy in Retinal Fundus Photographs," *JAMA*, vol. 316, no. 22, pp. 2402–2410, 2016.

[38] D. Gupta, B. Mood, J. Feigenbaum, K. R. B. Butler, and P. Traynor, "Using Intel Software Guard Extensions for Efficient Two-Party Secure Function Evaluation," in *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, 2016, pp. 302–318.

[39] C. Hazay, Y. Ishai, A. Marcedone, and M. Venkitasubramaniam, "LevioSA: Lightweight Secure Arithmetic Computation," in *Proceedings of the 2019 ACM Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, 2019, pp. 327–344.

[40] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778.

[41] ——, "Identity Mappings in Deep Residual Networks," in *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, 2016, pp. 630–645.

[42] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 772–783.

[43] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely Connected Convolutional Networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, 2017, pp. 2261–2269.

[44] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving Machine Learning as a Service," *CoRR*, vol. abs/1803.05961, 2018. [Online]. Available: https://arxiv.org/abs/1803.05961

[45] Intel, "Performance Considerations for Intel Software Guard Extensions (Intel SGX) Applications," 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/09/37/Intel-SGX-Performance-Considerations.pdf

[46] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, ser. JMLR Workshop and Conference Proceedings, vol. 37, 2015, pp. 448–456.

[47] M. Ion, B. Kreuter, A. E. Nergiz, S. Patel, M. Raykova, S. Saxena, K. Seth, D. Shanahan, and M. Yung, "On Deploying Secure Computing Commercially: Private Intersection-Sum Protocols and their Business Applications," Cryptology ePrint Archive, Report 2019/723, 2019, https://eprint.iacr.org/2019/723.

[48] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, 2018, pp. 2704–2713.

[49] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A Low Latency Framework for Secure Neural Network Inference," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, 2018, pp. 1651–1669.

[50] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang, "Optimizing Authenticated Garbling for Faster Secure Two-Party Computation," in *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, 2018, pp. 365–391.

[51] P. Koeberl, V. Phegade, A. Rajan, T. Schneider, S. Schulz, and M. Zhdanova, "Time to Rethink: Trust Brokerage Using Trusted Execution Environments," in *Trust and Trustworthy Computing - 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings*, 2015, pp. 181–190.

[52] V. Kolesnikov, M. Rosulek, N. Trieu, and X. Wang, "Scalable Private Set Union from Symmetric-Key Techniques," in *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part II*, 2019, pp. 636–666.

[53] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[54] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "ObliVM: A Programming Framework for Secure Computation," in *2015 IEEE Symposium on Security and Privacy, S&P 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 359–376.

[55] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious Neural Network Predictions via MiniONN Transformations," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 619–631.

[56] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - Secure Two-Party Computation System," in *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, 2004, pp. 287–302.

[57] L. Melis, C. Song, E. D. Cristofaro, and V. Shmatikov, "Exploiting Unintended Feature Leakage in Collaborative Learning," in *2019 IEEE Symposium on Security and Privacy, S&P 2019, San Francisco, CA, USA, May 19-23, 2019*, 2019, pp. 691–706.

[58] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A Cryptographic Inference Service for Neural Networks," in *29th USENIX Security Symposium, USENIX Security 20*, Boston, MA, 2020.

[59] J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman, "Information-Flow Control for Programming on Encrypted Data," in *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, 2012, pp. 45–60.

[60] P. Mohassel and P. Rindal, "ABY^3: A Mixed Protocol Framework for Machine Learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 35–52.

[61] P. Mohassel, M. Rosulek, and N. Trieu, "Practical Privacy-Preserving K-means Clustering," Cryptology ePrint Archive, Report 2019/1158, 2019, https://eprint.iacr.org/2019/1158.

[62] P. Mohassel and Y. Zhang, "SecureML: A System for Scalable Privacy-Preserving Machine Learning," in *2017 IEEE Symposium on Security and Privacy, S&P 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 19–38.

[63] B. Mood, D. Gupta, H. Carter, K. R. B. Butler, and P. Traynor, "Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, 2016, pp. 112–127.

[64] J. MSV, "Google's Research In Artificial Intelligence Helps In Preventing Blindness Caused by Diabetes," Forbes, sep 2017. [Online]. Available: https://www.forbes.com/sites/janakirammsv/2017/09/05/googles-research-in-artificial-intelligence-helps-in-preventing-blindness-caused-by-diabetes/#17b4f53256e1

[65] M. Nagel, M. van Baalen, T. Blankevoort, and M. Welling, "Data-Free Quantization through Weight Equalization and Bias Correction," *CoRR*, vol. abs/1906.04721, 2019. [Online]. Available: http://arxiv.org/abs/1906.04721

[66] J. D. Nielsen and M. I. Schwartzbach, "A domain-specific programming language for secure multiparty computation," in *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007*, 2007, pp. 21–30.

[67] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, "Privacy-preserving matrix factorization," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013, Berlin, Germany, November 4-8, 2013*, 2013, pp. 801–812.

[68] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-Preserving Ridge Regression on Hundreds of Millions

of Records," in *2013 IEEE Symposium on Security and Privacy, S&P 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 334–348.

[69] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 619–636.

[70] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations," in *2014 IEEE Symposium on Security and Privacy, S&P 2014, Berkeley, CA, USA, May 18-21, 2014*, 2014, pp. 655–670.

[71] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: XNOR-based Oblivious Deep Neural Network Inference," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, 2019, pp. 1501–1518.

[72] 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 *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, 2018, pp. 707–721.

[73] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, "A generic framework for privacy preserving deep learning," *CoRR*, vol. abs/1811.04017, 2018. [Online]. Available: http://arxiv.org/abs/1811.04017

[74] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in *2015 IEEE Symposium on Security and Privacy, S&P 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 38–54.

[75] J. So, B. Guler, A. S. Avestimehr, and P. Mohassel, "CodedPrivateML: A Fast and Privacy-Preserving Framework for Distributed Machine Learning," Cryptology ePrint Archive, Report 2019/140, 2019, https://eprint.iacr.org/2019/140.

[76] TensorFlow, "Broadcasting semantics," https://www.tensorflow.org/xla/broadcasting, 2018.

[77] F. Tramèr and D. Boneh, "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

[78] F. Tramèr, F. Zhang, H. Lin, J. Hubaux, A. Juels, and E. Shi, "Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge," in *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017, pp. 19–34.

[79] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-Party Secure Computation for Neural Network Training," *PoPETs*, vol. 2019, no. 3, pp. 26–49, 2019.

[80] X. Wang, S. Ranellucci, and J. Katz, "Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 21–37.

[81] ——, "Global-Scale Secure Multiparty Computation," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 39–56.

[82] A. C. Yao, "How to Generate and Exchange Secrets (Extended Abstract)," in *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. IEEE Computer Society, 1986, pp. 162–167.

[83] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, 2017, pp. 283–298.

[84] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Helen: Maliciously Secure Coopetitive Learning for Linear Models," in *2019 IEEE Symposium on Security and Privacy, S&P 2019, San Francisco, CA, USA, May 19-23, 2019*, 2019, pp. 724–738.

[85] X. Zhu, G. Iordanescu, I. Karmanov, and M. Zawaideh, Mar 2018. [Online]. Available: https://blogs.technet.microsoft.com/machinelearning/2018/03/07/using-microsoft-ai-to-build-a-lung-disease-prediction-model-using-chest-x-ray-images/

Appendix

A. Algorithms used by Porthos

The additional algorithms that reshape filters, input, and output, used by Porthos are shown in Algorithms 2, 3, and 4.

Algorithm 2 ReshapeFilter

```
Input: $X \in \mathbb{Z}_L^{f \times f}$.
Output: $Z \in \mathbb{Z}_L^{f^2 \times 1}$.
for $i = \{0, ..., f-1\}$ do
    for $j = \{0, ..., f-1\}$ do
        $Z[i \cdot f + j] = X[i][j]$
    end for
end for
```

Algorithm 3 ReshapeInput

```
Input: $X \in \mathbb{Z}_L^{m \times m}$.
Output: $Z \in \mathbb{Z}_L^{n^2 \times f^2}$ where $n = m - f + 1$.
Global Information: Filter dimension $f$.
for $i = \{0, ..., m-f\}$ do
    for $j = \{0, ..., m-f\}$ do
        for $k = \{0, ..., f-1\}$ do
            for $l = \{0, ..., f-1\}$ do
                $Z[i \cdot (m-f+1)+j][k \cdot f+j] = X[k+i][l+j]$
            end for
        end for
    end for
end for
```

Algorithm 4 ReshapeOutput

```
Input: $X \in \mathbb{Z}_L^{n^2 \times 1}$.
Output: $Z \in \mathbb{Z}_L^{n \times n}$.
for $i = \{0, ..., n-1\}$ do
    for $j = \{0, ..., n-1\}$ do
        $Z[i][j] = X[i \cdot n + j]$
    end for
end for
```

B. Batch Normalization

Batch Normalization [46] is used to normalize the inputs to intermediate layers across a mini-batch of images. For a batch B of inputs, let μ_B and σ_B^2 be the mean and the variance respectively. For an input x, the output of the batch normalization layer is defined as

$$BN(x) = \gamma \frac{(x - \mu_B)}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

where γ and β are the model parameters learned during training phase. In the inference phase, μ_B and σ_B^2 represent the mean and variance of the entire training dataset.

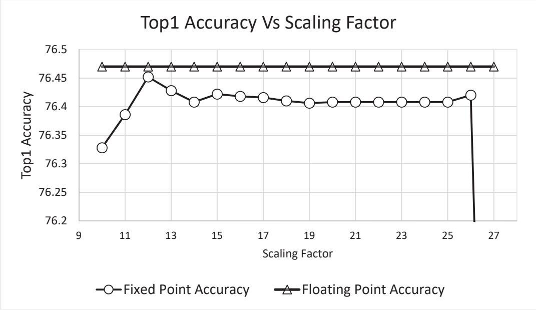


Fig. 11: ResNet50: Top 1 accuracy vs Scale

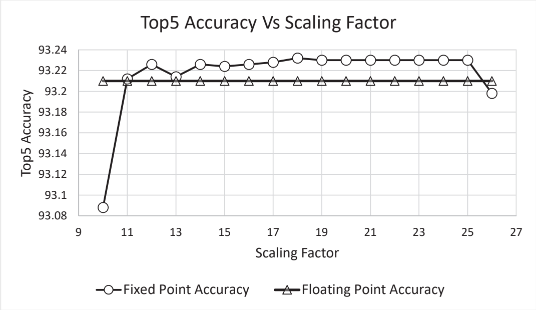


Fig. 12: ResNet50: Top 5 accuracy vs Scale

C. Accuracy of Athos

In this section, we present the Top 1 and Top 5 accuracies of Athos on the ImageNet dataset.

D. Comparison with 2PC/FHE

See Table X which validates the well-known fact that 3PC protocols like Porthos are much faster than 2PC/FHE-based approaches. We omit other 2PC/FHE works ([62], [19], [15], [14], [58], etc.) as the performance comparisons are similar and do not provide additional insights.

E. Proof of malicious security

For simplicity, consider the case of single malicious party P_i. Informally, we argue that our technique constrains P_i to follow the instructions of the semi-honest protocol $\pi(\cdot)$ faithfully. Or, deviating from faithful execution would result in some honest party to abort. The first Compute invocation of $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ fixes the input of P_i used in the protocol. Since every other $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_j,\mathsf{sk}_j)}$ reliably knows the verification key vk_i used by $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$, it checks the signatures on the function description (i.e., $\mathcal{T}_{\pi^*}^{(i)}$) as well as the messages of the protocol. The unforgeability of the signature scheme guarantees that P_i cannot forge signatures on incorrectly generated protocol messages. Note that we use this property to ensure that both of the following signatures cannot be forged: (a) signatures under vk_i on messages generated by $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ and sent to honest P_j (b) signatures under vk_j on messages sent by P_j being fed into $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$. Also, $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ provides correct randomness to generate messages of P_i in the semi-honest secure protocol. Hence, all messages from P_i to any honest party P_j are generated correctly as directed by π. This argument can be easily extended to multiple colluding corrupt parties.

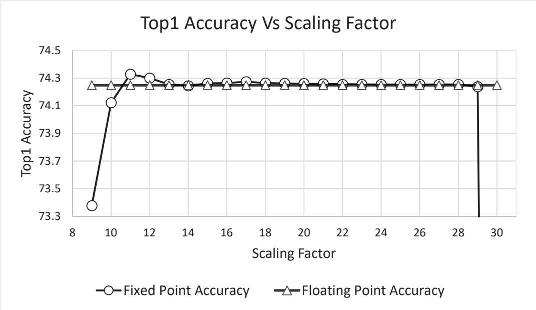


Fig. 13: DenseNet121: Top 1 accuracy vs Scale

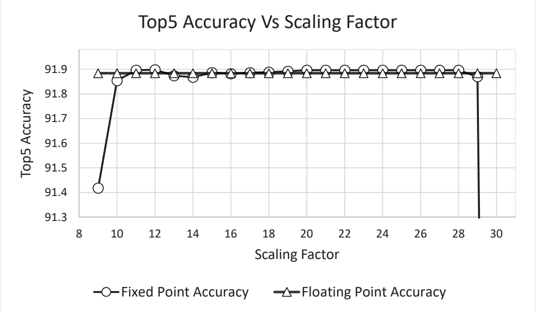


Fig. 14: DenseNet121: Top 5 accuracy vs Scale

Benchmark	CHET	MiniONN	Gazelle	Porthos
SqueezeNet* (CIFAR)	1342	-	-	0.05
MiniONN (CIFAR)	-	544	12.9	0.36
MiniONN (MNIST)	-	9.4	0.81	0.03

TABLE X: Comparison with 2PC – All times in seconds. CHET replaces ReLUs in a small SqueezeNet with square activations.

Formally, we give a security proof using the standard simulation paradigm (we refer the reader to [35], [18] for details on the paradigm). That is, the protocol in Figure 8 securely realizes the ideal MPC functionality described in Figure 9 against malicious adversaries.

Theorem 2 (Restated). *Let $\pi(\cdot)$ be a semi-honest secure MPC protocol securely realizing $\mathcal{F}_{mpc}^{f}$. Then, protocol $\mathsf{Prot}_{\mathsf{malicious}}(P_1, \cdots, P_n)$ described in Figure 8 securely realizes $\mathcal{F}_{mpc}^{f}$ in the $\mathcal{F}_{attest}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$−hybrid model (with $i \in [n]$) against malicious adversaries.*

Proof Sketch. Let $\mathcal{A}$ be the real world adversary. Ideal world adversary $\mathcal{S}$ that simulates the view of $\mathcal{A}$ is as follows: Let $\mathcal{S}'$ be the ideal world adversary or the semi-honest simulator for π (this exists because π is semi-honest secure). $\mathcal{S}$ picks $\{(\mathsf{vk}_k, \mathsf{sk}_k)\}_{k\in[n]}$ and gives $\{\mathsf{vk}_k\}_{k\in[n]}$ to $\mathcal{A}$. We denote a corrupt party by P_i and honest party by P_j. Next, when $\mathcal{A}$ invokes an instance of $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ on command Commit for a corrupted party P_i, $\mathcal{S}$ simulates the correct behavior of

$\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$. Also, $\mathcal{S}$ sends correctly generated tokens $\{\mathcal{T}_{\pi^*}^{(j)}\}$ for all honest parties to $\mathcal{A}$. When $\mathcal{S}$ receives token from $\mathcal{A}$ corresponding to a corrupted party P_i, it checks it against π^* and vk_i. It aborts if verification fails. When $\mathcal{A}$ invokes $\mathcal{F}_{\text{attest}}^{(\mathsf{vk}_i,\mathsf{sk}_i)}$ with x_i, $\mathcal{S}$ stores it as input of P_i. When $\mathcal{A}$ commits to inputs of all corrupt parties, $\mathcal{S}$ sends these to $\mathcal{F}_{\text{mpc}}^{f}$ to learn output y. It sends inputs of corrupt parties and outputs y to $\mathcal{S}'$ that generates the view of the adversary in the semi-honest protocol, that contains the randomness for all corrupt parties as well as the transcript of the protocol. Using this, it is easy for $\mathcal{S}$ to simulate the view of $\mathcal{A}$ in the rest of the protocol. The indistinguishability of the adversary's view in real and ideal executions follows from the semi-honest security of π. □