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 (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], ABY3 [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 [51]; (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
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 \(^2\). (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 \(^3\) have lower communication complexity and is more efficient than prior and concurrent works \(^4\). (Section V). Athos computes these Beaver triples \(^5\) 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 \(2qf^2 + 2f^2 + q^2\) elements in the ring \(\mathbb{Z}_{2^q}\), 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 \(2mf^2 + q^2\) 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).

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

\(1\)https://github.com/mpc-msri/EzPC

\(2\)ABY\(^6\) 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.

**Porthos (Section IV)**. Porthos is an improved semi-honest 3-party secure computation protocol (tolerating one corruption) that builds upon SecureNN \(^7\). Porthos makes two crucial modifications to SecureNN. First, SecureNN reduces convolutions to matrix multiplications and invokes the Beaver triples \(^8\) 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 \(2qf^2 + 2f^2 + q^2\) elements in the ring \(\mathbb{Z}_{2^q}\), 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 \(2mf^2 + q^2\) 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
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], [13], [77], [32], [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) [5] 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], [71]), 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 and conclude in Section VII.

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 I. We describe how code compilation happens from TensorFlow to MPC protocols. 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
Athos front-end and HLIL

Athos front-end compiles the input TensorFlow models to HLIL (described next) with explicit tensor dimensions. To obtain these dimensions, the front-end 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 front-end. 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 to ensure this 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 front-end passes the broadcasting information to HLIL, which then accounts for it by compiling it to the appropriate LLIL library function call.

Below we explain each of these components in detail.

---

A. Frontend and HLIL

Athos front-end compiles the input TensorFlow models to HLIL (described next) with explicit tensor dimensions. To obtain these dimensions, the front-end 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 front-end. 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 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” 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 front-end passes the broadcasting information to HLIL, which then accounts for it by compiling it to the appropriate LLIL library function call.

Below we explain each of these components in detail.

---

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 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 $\rho_s$ to matrices $M$ of Reals where the result is a point-wise application of $\rho_s$ 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):

\[
F(\text{MatAdd}(A,B,C)) = \text{MatAdd}(A,B,C) \\
F(\text{MatMul}(A,B,C)) = \text{MatMul}(A,B,C); \\
\text{ScaleDown}(C,s)
\]

As an example of the conversion process, consider the program $M_1 \ast M_2$ 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 $400.1 \ast 0.3 + 200.1 \ast 0.1$ is 140.04. Single-precision floating-point arithmetic with 32 bits only has a 23-bit mantissa and computes the approximately correct result 140.040099. We use 0.1f to denote the floating-point number closest to the Real number 0.1. Given $s = 24$, $F(M_1 \ast M_2)$ results in the following program over integers

\[
(\rho_{24}(400.1f) \ast \rho_{24}(0.3f) + \rho_{24}(200.1f) \ast \rho_{24}(0.1f)) >> 24
\]

which results in the following computation with 64-bit integers

\[
(6712564224 \ast 3357121024 + 5033165 \ast 1677721) >> 24
\]

The final result is 2349481329 which represents the real number $2349481329 = 140.040000021457672119140625$ which is good approximation of the desired result 140.04. Although it is feasible to construct 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 $F([400.1f, 200.1f] \ast [0.3f, 0.1f])$ is $(1600 \ast 1 + 800 \ast 0) >> 24$, 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.

Next, suppose $s$ is set to a very high value, e.g., 60. Then, the computation $\rho_{60}(400.1f) \ast \rho_{60}(0.3f)$ 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 $400.1f \ast 0.3f + 200.1f \ast 0.1f$, 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

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 (\( \text{extern } d \)) 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 LLIL 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(\( \cdot \))) idiom. For protocols like Porthos and SecureNN [79] that reduce ReLU and MaxPool to secure comparison protocols, ReLU(MaxPool(\( \cdot \))) can be much more efficient than MaxPool(ReLU(\( \cdot \))) 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 LLIL that replaces occurrences of MaxPool(a, b, ReLU(A)); with ReLU(MaxPool(a, b, A)). For example, if the input matrix \( A \) has dimensions 112 × 112 × 64 and we compute a MaxPool with 2 × 2 windows. Then, the output matrix has dimensions 56 × 56 × 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 IV).

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 \( \rho \) 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 \texttt{main} and for each call \texttt{ScaleDown}(A, 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 \( \rho \). 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 \textit{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:

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

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.
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 $m \times n$ 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 $(x)_i$ and $(x)_i'$ denote the two shares of a 2-out-of-2 additive secret sharing of $x$ over $Z_n$ – in more detail, pick $r \leftarrow Z_n$, set $(x)_0 = r$ and $(x)_1 = x - r \pmod{t}$. $(x)_i'$ denotes a sharing of $x$ over $Z_n$. Reconstruction of a value $x$ from its shares $(x)_0$ and $(x)_1$ is simply $(x)_0 + (x)_1$ over $Z_n$. This generalizes to larger dimensions - e.g. for the $m \times n$ matrix $X$, $(X)_i$ and $(X)_i'$ denote the matrices that are created by secret sharing the elements of $X$ component-wise (other matrix notation such as $\text{Recon}^t(X_0, X_1)$ are similarly defined).

Let Conv$_{2d,m,f}$ 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 Conv$_{2d,m,f}$ is described in Algorithm 1 where $L = 2^f$, $\ell = 64$. Algorithms ReshapeInput, ReshapeFilter and 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 $Z_n$. That is, $P_j$ holds $(X)_i^j, (Y)_i^j$ for $j \in \{0, 1\}$. In SecureNN, $P_0$ first reshapes $(X)_i^0$ into $(X)_i^0'$ by running ReshapeInput. Then, it picks a random matrix $(A)_i^0'$ of same size as $X'$ and sends $(E')_i^0 = (X)_i^0 - (A)_i^0$ to $P_1$ that requires communicating $q^2 f^2$ elements. In Porthos, we optimize this as follows: $P_0$ picks a random matrix $(A)_i^0$ of same size as $X$ (Step 1) and sends $(E)_i^0 = (X)_i^0 - (A)_i^0$ 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 $(2q^2 f^2 + 2f^2 + q^2 f)\ell$ in SecureNN to $(2m^2 + 2f^2 + q^2 f)\ell$. 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 Conv$_{2d,m,f}$

**Input:** $P_0$ holds $((X)_i^0, (Y)_i^0)$ and $P_1$ holds $((X)_i^1, (Y)_i^1)$, where $X \in Z_n^{m \times m}$, $Y \in Z_n^{f \times f}$.

**Output:** $P_0$ gets $\langle \text{Conv}_{2d,m,f}(X,Y)\rangle^0$ and $P_1$ gets $\langle \text{Conv}_{2d,m,f}(X,Y)\rangle^1$.

**Common Randomness:** $P_0 \& P_1$ hold shares of a zero matrix $U$ of dimension $q \times q$, $q = m - f + 1$. $P_0 \& P_1$ hold a common PRF key $k_0$, and $P_0 \& P_1$ hold a common PRF key $k_1$.

1. $P_0 \& P_2$ use PRF key $k_0$ to generate random matrices $(A)_i^0 \in \mathbb{Z}_{n}^{m \times m}$, $(B)_i^0 \in \mathbb{Z}_{n}^{f \times f}$ and $(C)_i^0 \in \mathbb{Z}_{n}^{q \times q}$.
2. $P_2$ and $P_0$ use PRF key $k_1$ to generate random matrices $(A)_i^1 \in \mathbb{Z}_{n}^{m \times m}$ and $(B)_i^1 \in \mathbb{Z}_{n}^{f \times f}$.
3. $P_2$ computes $A = (A)_i^0 + (A)_i^1$ and $B = (B)_i^0 + (B)_i^1$.
4. Let $A' = \text{Reshapelnput}(A)$ and $B' = \text{ReshapeFilter}(B)$.
5. $P_2$ computes $C = A' \cdot B' - (C)_i^0$ and sends it to $P_1$.
6. For $j \in \{0, 1\}$, $P_j$ computes $(E')_i^j = (X)_i^j - (A)_i^j$ and $(F)_i^j = (Y)_i^j - (B)_i^j$ and sends to $P_{j+1}$.
7. $P_0 \& P_1$ reconstruct $E$ and $F$ using exchanged shares.
8. For $j \in \{0, 1\}$, $P_j$ computes $(X)_i^j = \text{Reshapelnput}(X)_i^j$, $E''_j = \text{Reshapelnput}(E)$, $(Y')_i^j = \text{ReshapeFilter}(Y)_i^j$, $F' = \text{ReshapeFilter}(F)$.
9. For $j \in \{0, 1\}$, $P_j$ computes $(Z')_i^j = -E''_j \cdot F' + (X)_i^j \cdot F'' + E''_j \cdot (Y')_i^j + (C)_i^0 + (U)_i^j$.
10. For $j \in \{0, 1\}$, $P_j$ outputs $(Z)_i^j = \text{Reshapelnput}(Z')_i^j$.

**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 $L = 2^d$. For both of these, parties run a protocol called ComputeMSB to evaluate most significant bit (MSB) of secret values. This protocol require shares over $L - 1$. So parties run a protocol called ShareConvert to convert shares over $L$ to shares over $L - 1$. Both protocols ComputeMSB and 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
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 ShareConvert, ComputeMSB, ReLU and MaxPool by 25%.

The revised table with comparison of overall communication complexity of all protocols with improvements over SecureNN are provided in Table III. 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. MaxPool_{o} 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).

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Communication (SecureNN)</th>
<th>Communication (Porthos)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv2d_{m,i,f,o}</td>
<td>$(2^{p}f^2i + 2^{p}f^2o + q^2o)\ell$</td>
<td>$(2^{m}f^2i + 2^{m}f^2o + q^2o)\ell$</td>
</tr>
<tr>
<td>ShareConvert</td>
<td>$4\ell \log p + 6\ell$</td>
<td>$4\ell \log p + 5\ell$</td>
</tr>
<tr>
<td>ComputeMSB</td>
<td>$4\ell \log p + 13\ell$</td>
<td>$4\ell \log p + 9\ell$</td>
</tr>
<tr>
<td>ReLU</td>
<td>$8\ell \log p + 24\ell$</td>
<td>$6\ell \log p + 19\ell$</td>
</tr>
<tr>
<td>MaxPool_{o}</td>
<td>$(6\ell \log p + 22\ell)(n - 1)$</td>
<td>$(6\ell \log p + 24\ell)(n - 1)$</td>
</tr>
</tbody>
</table>

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

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], [33], [11], [23], [29], [51], [52], [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 $P_i$ can obtain guarantees on the correctness of protocol messages sent by $P_j$ as long as these messages are computed and signed by $P_j$’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 $F_{\text{attest}}$

Description. We formally define the ideal functionality for attested executions in Figure 7. The functionality is parameterized by a signing key pair $(vk, sk)$. Let $\text{Sign}_{sk}(m)$ denote the signing algorithm on message $m$ and $\text{Verify}_{vk}(m, \sigma)$ denote verification of signature $\sigma$ 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 $T_g$ generated as $\text{Sign}_{sk}(H(g))$, where $H$ is a collision resistant hash function. Note that this token is publicly verifiable given $g$ and $vk$. Let $state_{ctr}$ 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 $F_{\text{attest}}$ is initialized, the initial state $state_0$ is empty (or, $\epsilon$). Subsequent invocations of the functionality is done on input $w_{\text{ctr}}$ using the Compute command. The function $g$ is a deterministic mapping from $(ctr, w_{\text{ctr}}, r_{\text{ctr}}, state_{ctr-1})$ to $(y_{\text{ctr}}, state_{ctr})$, where $r_{\text{ctr}}$ is the required randomness. The functionality picks randomness $r_{\text{ctr}}$, evaluates $g$ and provide a signature on the function output $y_{\text{ctr}}$ using the signing key sk. Furthermore, $(y_{\text{ctr}}, state_{ctr})$ is always given to party $P$ such that $state_{ctr}$ contains $r_{\text{ctr}}$ 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 $g, w_{\text{ctr}}, r_{\text{ctr}}, ctr, state_{ctr}$, 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 $F^{(vk, sk)}_{\text{state}}$

$F_{\text{state}}$ interacts with a party $P_i$.
- On input message $(\text{Commit}, g)$ from $P_i$,
  1) Record $(\text{Commit}, \text{state}_0)$, where $\text{state}_0 = e$;
  2) Send $(\text{state}_0, T_0)$ to $P_i$, where $T_0 = \text{Sign}_u(H(g))$.
  3) Ignore further $(\text{Commit}, g)$ messages.
- On input message $(\text{Compute}, w_{cr}^r)$ from $P_i$, retrieve $\text{state}_{cr-1}$ pick required randomness $r_{cr}$ and compute $(g_{\text{cr}}, u_{cr}, r_{cr}, \text{state}_{cr-1})$ to obtain $(y_{\text{cr}}, \text{state}_{cr})$ such that $\text{state}_{cr}$ contains $(y_{\text{cr}}, r_{\text{cr}})$. Send $(y_{\text{cr}}, T_0 = \text{Sign}_u(y_{\text{cr}}||\text{cr}), \text{state}_{cr})$ to $P_i$.

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

the $F^{(vk, sk)}_{\text{state}}$-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 $F^{(vk_i, sk_j)}_{\text{state}}$ functionality interacting with $P_i$, where $(vk_i, sk_j)$ are keys used by the functionality. These messages can be verified by functionality $F^{(vk_i, sk_j)}_{\text{state}}$ interacting with $P_j$. We assume that every $F^{(vk_i, sk_j)}_{\text{state}}$ knows the verification key $vk_j$ used by functionalities of all other parties $P_i$ 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 $r_{i,j,\text{ctr}}$ and Transcript$_i$, which includes the transcript of all messages sent and received by the party $P_i$ so far. Given these, $\pi(\cdot)$ outputs $y_{i,j}^{\text{ctr}}$, which is the message that $P_i$ must send to $P_j$ in round $\text{ctr}$ and also updates Transcript$_i$, appropriately. Additionally, $\pi(\cdot)$ takes message $y_{i,j}^{\text{ctr}}$ sent by $P_j$ to $P_i$ at round $\text{ctr}$ and update 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 $\text{ctr} = 1$, Let $x_i$ be the input of $P_i$ in $\pi(\cdot)$. Then, $(\text{ctr}, x_i)$ is stored as state$_1$ (also called as Transcript$_1^i$) and sent to $P_i$.
2) When $\pi^*(\cdot)$ receives a message $M = (y_{i,j}^{\text{ctr}}, \text{ctr}, \sigma)$ from party $P_j$, it runs $\text{Verify}_{vk_j}((y_{i,j}^{\text{ctr}}, \text{ctr}, \sigma))$. If verification succeeds, it appends $M$ to Transcript$_i$. Else, $P_i$ aborts.
3) $\pi^*(\cdot)$ on input $(\text{ctr}, \text{state}_{\text{ctr}-1}, j)$ computes the next message from $P_i$ to $P_j$ as follows: It checks that state$_{\text{ctr}-1}$ contains a valid transcript of all messages computed so far. If it verifies, it picks randomness $r_{i,j,\text{ctr}}$ and runs $\pi(\text{ctr}, \text{state}_{\text{ctr}-1, j}, r_{i,j,\text{ctr}})$ to compute next message $y_{i,j}^{\text{ctr}}$ and updated state state$_{\text{ctr}}$ (containing $r_{i,j,\text{ctr}}$). Else it outputs $\bot$. Note that state$_{\text{ctr}-1}$ already contains input $x_j$, the input of party $P_j$.

Malicious MPC in the $F^{(vk, sk)}_{\text{state}}$-hybrid model. The malicious MPC protocol works as follows: Each party $P_i$ invokes $F^{(vk_i, sk_i)}_{\text{state}}$ with the command Commit using function $\pi^*(\cdot)$ described above and sends the received token $T_{\pi^*}$ to other parties $P_j$. It receives similar tokens $T_{\pi^*j}$ 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 $F^{(vk, sk)}_{\text{state}}$ as described formally in Figure 8.

Functionality $F^{\text{malicious}}_{\text{prot}}(P_1, \cdots, P_n)$

$F^{\text{malicious}}_{\text{prot}}$ interacts with parties $\{P_i, \cdots, P_n\}$ and the adversary $S$.
- 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 $S$.
- Upon receiving $(i, \text{Send})$ or $(i, \bot)$ from $S$, send $y$ or $\bot$ to $P_i$.

Fig. 8: Malicious secure MPC $F^{\text{malicious}}_{\text{prot}}(P_1, \cdots, P_n)$.

Malicious Security. Next, we prove that if $\pi$ 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 $F^{\text{malicious}}_{\text{prot}}$. Then, protocol $F^{\text{malicious}}_{\text{prot}}(P_1, \cdots, P_n)$ described in Figure 8 securely realizes $F^{\text{malicious}}_{\text{prot}}$ in the $F^{(vk, sk)}_{\text{state}}$-hybrid model (with $i \in \{n\}$) against malicious adversaries.
C. Realizing $F_{\text{attr}}$

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 $r_{ct}$ of Figure 7 can be sampled in SGX using $\text{sgx_read_rand}$ command. The attestation token $T_g$ 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 $(vk, sk)$ 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 $T_g$ contains the verification key $vk$ in the clear and this $vk$ can be used to verify the signed outputs $y_{\text{test}}$. Now, on receiving the Compute command, the enclave starts executing the code of $g$ and produces outputs signed under $sk$.

While running MPC in the $F_{\text{attr}}$-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 $\pi^*$ (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 $\pi^*$ (appended with this prelude), they can compute the hash of $\pi^*$ 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. 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 II-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, 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 CryptTFLOW to securely compute inference on the ImageNet dataset using the following TensorFlow programs: ResNet50 and DenseNet121. We also show that the performance of semi-honest and malicious protocols generated by CryptTFLOW scale linearly with the depth of DNNs. Second, in Section VI-B, we show that CryptTFLOW outperforms prior works on secure inference of DNNs. Next, we evaluate each component of CryptTFLOW 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 and Porthos. We show Aramis-based malicious secure inference outperforms pure crypto-based malicious secure protocols by huge margins in Section VI-E. Finally, in section VI-F, we discuss two case-studies of running CryptTFLOW on DNNs for healthcare. We begin by providing details of the systems

https://github.com/tensorflow/models/tree/master/official/r1/resnet
https://github.com/pudae/tensorflow-densenet
TABLE IV: CRYPTFLOW: ImageNet scale benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Semi-Honest (s)</th>
<th>Malicious (s)</th>
<th>Comm. (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESNET50</td>
<td>25.9</td>
<td>75.4</td>
<td>6.9</td>
</tr>
<tr>
<td>DENSENET121</td>
<td>36.0</td>
<td>112.9</td>
<td>10.5</td>
</tr>
</tbody>
</table>

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. 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\(^4\) [12], 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 [V]. 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

5Specifically, 11, 22, 25, 44 and 60 million parameters for RESNET-\(n\) for \(n = 18, 34, 50, 101,\) and 152 respectively.

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

**Modularity.** Since **CryptTFLOW** 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 **CryptTFLOW**. 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 **CryptTFLOW** 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 VI. Additionally, we also compare Porthos and SecureNN on ImageNet scale benchmarks in Table VII. For this purpose, we add the code of SecureNN available at [5] as another backend to **CryptTFLOW**. 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.

**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 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 [V] 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 **CryptTFLOW** 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 **CryptTFLOW**.

**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], [87]. Predicting whether a retina image has DR or not can be done securely in about 30 seconds with **CryptTFLOW**.

**VII. RELATED WORK**

**High level languages.** **CryptTFLOW** 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], ObiVM [54], CBMC-GC [42], SMCL [60], 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 CRYSTAL [42] and Delphi [58], our more ambitious goal is to extend CRYSTAL to support TensorFlow. The research literature in float-to-fixed protocols is better.

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], [57], [72]), or by using non-standard training algorithms that output fixed-point models [71], [71]. 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 losses [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 [50], [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 CRYSTAL is better.

Secure Machine Learning. There has been a flurry of recent results [61], [75], [25], [44] in the area of secure machine learning, both in the 2-party [77], [67], [34], [55], [49], [57], [84], as well as in the 3-party setting [72], [60], [79], [12]. The most relevant to our work are ABY [3] and SecureNN [19] 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. XONNN [71] requires DNNs to have binary weights, etc. On the other hand, we focus on standard inference algorithms and CRYSTAL 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

CRYSTAL 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 CRYSTAL, 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). CRYSTAL’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 CRYSTAL. Our more ambitious goal is to extend CRYSTAL 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.


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 $F_{\text{act}}(vk_i, sk_i)$ fixes the input of $P_i$ used in the protocol. Since every other $F_{\text{act}}(vk_j, sk_j)$ reliably knows the verification key $vk_i$ used by $F_{\text{act}}(vk_i, sk_i)$, it checks the signatures on the function description (i.e., $T_{\pi(i)}$) 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 $F_{\text{act}}(vk_i, sk_i)$ and sent to honest $P_j$ (b) signatures under $vk_j$ on messages sent by $P_j$ being fed into $F_{\text{act}}(vk_i, sk_i)$. Also, $F_{\text{act}}(vk_i, sk_i)$ 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 $\pi$. This argument can be easily extended to multiple colluding corrupt parties.

Formally, we give a security proof using the standard simulation paradigm (we refer the reader to [35], [13] 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 $F_{\text{act}}$. Then, protocol $\text{Prot}_{\text{malicious}}(P_1, \ldots, P_n)$ described in Figure 8 securely realizes $F_{\text{act}}$ in the $F_{\text{act}}(vk_i, sk_i)$-hybrid model (with $i \in [n]$) against malicious adversaries.

**Proof Sketch.** Let $A$ be the real world adversary. Ideal world adversary $S$ that simulates the view of $A$ is as follows: Let $S'$ be the ideal world adversary or the semi-honest simulator for $\pi$ (this exists because $\pi$ is semi-honest secure). $S$ picks $\{vk_k, sk_k\}_{k \in [n]}$ and gives $\{vk_k\}_{k \in [n]}$ to $A$. We denote a corrupt party by $P_i$ and honest party by $P_j$. Next, when $A$ invokes an instance of $F_{\text{act}}(vk_i, sk_i)$ on command Commit for a corrupted party $P_i$, $S$ simulates the correct behavior of...
\( F^{(vk_i, sk_i)} \). Also, \( S \) sends correctly generated tokens \( \{ T^{(j)}_{\pi^*} \} \) for all honest parties to \( \mathcal{A} \). When \( S \) receives token from \( \mathcal{A} \) corresponding to a corrupted party \( P_i \), it checks it against \( \pi^* \) and \( vk_i \). It aborts if verification fails. When \( \mathcal{A} \) invokes \( F^{(vk_i, sk_i)} \) with \( x_i \), \( S \) stores it as input of \( P_i \). When \( \mathcal{A} \) commits to inputs of all corrupt parties, \( S \) sends these to \( F^{I_{mpc}} \) to learn output \( y \). It sends inputs of corrupt parties and outputs \( y \) to \( S' \) 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 \( S \) 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 \( \pi \). \( \square \)