TONIC: Towards Oblivious Neural Inference Compiler

Po-Hsuan Huang
Department of Computer Science and Information Engineering, National Cheng Kung University
Tainan, Taiwan
aben20807@gmail.com

Chia-Heng Tu
Department of Computer Science and Information Engineering, National Cheng Kung University
Tainan, Taiwan
chiaheng@mail.ncku.edu.tw

Shen-Ming Chung
Industrial Technology Research Institute
Hsinchu, Taiwan
antonius@itri.org.tw

ABSTRACT

Privacy-preserving deep learning computing becomes popular these days as it helps protect, for example, both user data and deep neural network (DNN) model parameters at the same time with cryptographic techniques. In particular, significant efforts have been made to leverage secure two-party computation schemes for preventing user/model data from disclosure during DNN inference. Nevertheless, the existing works require manual intervention while converting trained models into secure computation programs, which is not scalable to modern deep networks efficiently. In this work, we propose a compiler framework, TONIC, to do the conversion automatically with scalability. Given a pre-trained DNN model, TONIC converts it into one of two secure two-party computation languages, i.e., ObliVM and ABY. Based on tailored backends built on top of a DNN compiler, TVM, our case studies show that TONIC is able to automatically convert popular DNN models, such as CryptoNets and MobileNetV2, into the corresponding programs for secure computations.

CCS CONCEPTS

• **Computer systems organization** → *Neural networks*; • **Software and its engineering** → **Source code generation**; • **Security and privacy** → **Software and application security**;

KEYWORDS

Privacy-preserving inference; deep neural networks; deep neural network compilation; secure two-party computation

ACM Reference Format:
Po-Hsuan Huang, Chia-Heng Tu, and Shen-Ming Chung. 2021. TONIC: Towards Oblivious Neural Inference Compiler. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3441929

1 INTRODUCTION

As deep learning (DL) techniques become pervasive in various application domains, data privacy concerns are raised since DL requires user data to improve its knowledge base while the data privacy could be compromised. This phenomenon exhibits the self-contradiction challenge for the advancements of DL technologies. One promising solution to the contradiction situation is known as *secure multiparty computation (SMPC)* (or secure function evaluation, SFE), which provides privacy-preserving computations by allowing participants to do a computation cooperatively over their data while keeping the data private to each individual participant, where each of the participants has access to the computation results. As a result, SMPC protects the data privacy of participants from each other.

The concept of SMPC originated from the mental poker work published in 1979 [32], and secure two-party computation (2PC) was achieved by the garbled circuit (GC) protocol introduced in 1982 [34], known as the Millionaires' Problem, where two millionaires want to know which of them is richer without disclosing their actual wealth status. The two-party model was generalized into the multi-party model by the GMW sharing scheme [19] in 1987, which presented the basic scheme for the following essential secure multi-party protocols. For instance, oblivious transfer (OT) protocol [22] was shown to be useful for a data sender and a receiver to pass the data, where the sender transfers one of many pieces of information to the receiver, but it is oblivious to the transferred data contents. OT is considered to be complete for SMPC.

Oblivious computation refers to an OT-based protocol for 2PC, which is free from either direct or indirect data content leaks, and the oblivious computation based solutions have been proposed for performing secure DL inferences. For instance, considering a trained DL model used for a medical disease diagnosis application, there is a patient holding his/her medical information on the client machine and a server possessing the DL model for the diagnosis task; the oblivious DL prevents the leaking of the plaintext medical data and the DL model (e.g., the parameters) from each other by running the privacy-preserving inference operations simultaneously on both the client and server sides, computing on the encrypted data and exchanging data over the secrete channel (i.e., OT), which is referred to as *oblivious neural inference* in this work.

Oblivious neural inference solutions (e.g., DeepSecure [29], SecureML [25], MiniONN [24], Gazelle [21], HyCC [7], EzPC [8], and XONN [27]) have been developed to facilitate the secure computation by utilizing one or more cryptographic tools, such as GC and GMW. In particular, EzPC and XONN are the latest efforts that require their users to define the deep neural networks and generate the corresponding programs using the cryptographic protocols, i.e., EzPC using Boolean and Arithmetic circuits, whereas XONN adopting GC circuits. Nevertheless, the above works are not suitable for some occasions; for example, in the above medical application,

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3412841.3441929

it would require a tool to automatically convert the incremental learned DL model into the oblivious DL version in a timely manner. In such a case, it is a time-consuming job to use their proposed methods to train the built model (i.e., the converted incremental DL model) again whenever the incremental DL model is updated. Furthermore, the above works are not shown they are scalable to a deep network architecture (e.g., MobileNetV2 [30] with 52 convolutional layers, 35 clips, and 1 fully-connected layer), which makes them hard to be applied for real-world applications that require a more sophisticated architecture.

In this work, we propose a compiler framework, TONIC, for converting a trained DL model into its 2PC version automatically for secure computing, so that the client is able to protect its user data privacy and the server can keep its model parameters private to itself. The software architecture of TONIC is illustrated in Figure 1. Thanks to TVM, an open-source graph compiler converting a model trained by various DL frameworks into machine executables, TONIC is able to accept the pre-trained DL models. In the current setup, TONIC supports two kinds of backends that convert a trained model into its 2PC versions (i.e., ObliVMLang and ABY programs), where ObliVMLang is a high-level Java-based language for writing 2PC programs, and ABY provides the lower-level C++ APIs to develop 2PC programs; more detailed information about ObliVM [23] and ABY [16] is given in Section 2. TONIC is able to provide oblivious DL inferences for a variety of models for MNIST and ImageNet datasets. In particular, TONIC is able to convert the MobileNetV2 model into its 2PC version with the ABY backend and encouraging results are obtained.

In the rest of this paper, the background information of the involved software frameworks is introduced in Section 2. The TONIC framework is described in Section 3. Section 4 compares the delivered performance between TONIC and prior work. The related work is introduced in Section 5. Section 6 concludes this work.

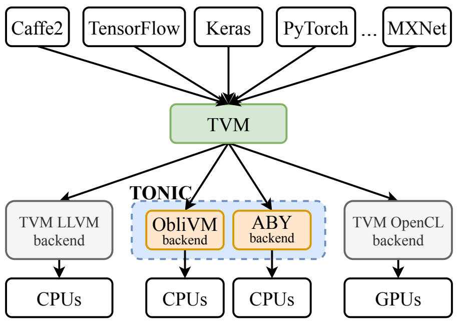

Figure 1: Overview of TONIC software architecture.

2 BACKGROUND

The open-source software projects involved in this work are introduced in the following three subsections, which is followed by the introduction of the system model for oblivious neural inference.

2.1 TVM

DL frameworks (e.g., TensorFlow and PyTorch) have been used to facilitate the development of DL-based applications by exposing high-level interfaces to application developers, and have been designed to allow developed models to be migrated from one computer platform to another for easier deployments (i.e., from the private server to a cloud server). A graph compiler like TVM [10] serves the purpose of optimizing the performance of DL models, which are generated by different DL frameworks, on a variety of hardware targets. The optimizations are done by the graph compiler backends, each of which is responsible for optimizing the performance of the given DL model (that is represented as a computation graph within the compiler) for a specific hardware architecture. As illustrated in Figure 1, TVM is capable of generating the codes to run on CPU and GPU, respectively.

TONIC backends developed in this work are a little bit different from the conventional backends mentioned above since the two backends generate the programs in the specific 2PC languages (i.e., ObliVMLang and ABY, which are further introduced in the following subsections), which require further conversions using cryptographic protocols before they are compiled into the CPU executables. In other words, the two backends are more like the source-to-source compilers (i.e., translating DL models into the two 2PC languages), instead of a typical compiler backend generating a lower level code specialized for target hardware architecture.

2.2 ObliVM

ObliVM [23] provides a high-level language-based framework for SMPC using the GC cryptographic protocol. ObliVM introduces the Java-based domain-specific language and uses *ObliVMLang* to compile ObliVM code written by application developers into 2PC program in Java, which is an ordinary Java program and can be further converted into Java bytecode run by Java virtual machine. During the 2PC programs' execution, the garbled circuit library, *ObliVMGC*, is invoked by the 2PC programs whenever necessary to generate the garbled circuits emulating the behaviors defined in the ObliVM program. Note that application developers write a single ObliVM program, which will be converted into the corresponding Java program for 2PC, where the two participants use the argument list to configure the *party* for each of the two Java program instances, which run simultaneously and collaboratively to perform the works defined in the ObliVM program.

In ObliVM, garbled circuits are generated on-the-fly (i.e., generating the garbled circuits for the required operations as needed right before its execution), and this design is able to handle the application with large circuits since it is able to reduce the peak memory footprint consumed by the circuits, compared with the method which has to generate the entire garbled circuits for a code segment before the code execution. In addition, ObliVM leverages the garbage collection feature offered by Java virtual machine, and the application developers do not have to worry about the memory management issue (i.e., they do not have to manage the memory explicitly in the code). Another advantage of ObliVM is that it is Java-based software, which allows it to run on any platforms that can run the Java virtual machine.

2.3 ABY

ABY [16] is a C++ language extension to support secure 2PC with its library. Unlike ObliVM, which supports the GC protocol only, ABY has the implementation of three cryptographic protocols: Arithmetic sharing (A), Boolean sharing (B), and Yao's GC (Y), and it provides conversion mechanisms between each two of the three protocols. Each protocol has its own advantage over the other two, i.e., Arithmetic sharing doing well for additions and multiplications, Boolean sharing performing better in multiplexer situations, and GC for comparison operations. By allowing to switch among the three different protocols, ABY makes the 2PC more applicable for different application domains. Because the mechanism of Boolean sharing and Arithmetic sharing are similar, we group them as the GMW-based protocols later.

While one can use either ObliVM or ABY to implement the same application for 2PC execution, they are very different in the following aspects: 1) the programming abstraction level, 2) the supported cryptographic protocols, and 3) the execution scheme, as well as 4) the execution efficiency. Different from the high-level programming language of ObliVM, which requires very little knowledge about multi-party computation (i.e., the GC protocol) to write an ObliVM program, ABY users need to use the ABY APIs to perform the arithmetic operations for the computations required by the target application and to switch among the three protocols with explicit ABY function calls so as to enhance the execution efficiency. In contrast to the dynamic circuit generation scheme in ObliVM, ABY adopts the setup-and-execution scheme, where the garbled circuits are able to run only after they are set up properly. ABY users are responsible for taking care of the memory footprints required by the arranged circuits in order to prevent memory overflows caused by excessive circuits used in the setup phase. To sum up, it is easier to program 2PC computation with the Java-like language with ObliVM, but ObliVM runs on top of the Java virtual machine, which incurs some level of overhead when emulating the execution of garbled circuits within the virtual machine. On the contrary, while ABY requires extensive knowledge to program 2PC applications, the primitive cryptographic protocol functions are more efficient for the execution. Section 4 further compares their delivered performance.

2.4 System Model for Oblivious Neural Inference

A 2PC system for deep learning inferences is composed of two participants, where one is the user data provider and the other is the model provider. The system requires data from both participants to accomplish the required model inference. Usually, the model provider runs a service (e.g., deep learning as a service) to take the user data as the input to its pre-built DL model and to output the prediction results regarding the given input. Since it is often the case that the trained DL model parameters are way more valuable than the DL network architecture, it is a natural design decision to express the neural network architectures as the 2PC programs, where the user data and the model parameters are fed to the two individual program instances, respectively. In such a case, the above model is able to protect their private data from each other.

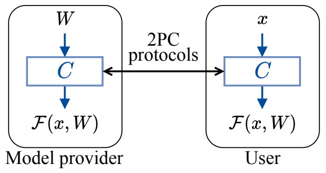

Figure 2: The system model for oblivious neural inference, where there are two participants, the model provider and the user, and both run the 2PC programs for the neural network inference operations of the pre-trained DL model. Note that x refers to user private data, W is the pre-trained weights of the DL model, whose mathematical operations are performed by the functions $\mathcal{F}$ and C represents the corresponding circuits of $\mathcal{F}$. For further information about protocols, please refer to ABY [16].

Figure 2 illustrates the above system model for oblivious neural inference. Specifically, the model provider runs its own share of the 2PC program and takes the parameters of the model as input, where the corresponding network architecture and operations of a pre-trained DL model are defined in the 2PC program. On the other hand, the user also runs its share of the 2PC program in parallel to the model provider, and the user's program is fed with the data private to the user. When there is a need for data exchanges, the 2PC protocol(s) is used to secretly transferred the required data between the 2PC programs. Finally, both parties are able to obtain the prediction results at the end of the program executions.

3 OBLIVIOUS NEURAL INFERENCE COMPILER

For automatically compiling the pre-trained DL models, especially for convolutional neural networks (CNNs) produced by the DL software frameworks, TVM is used and served as the compiler frontend to accept the existing, trained models and to convert the models into its intermediate representation (i.e., TVM IR). Our proposed oblivious neural inference compiler is actually built upon the TVM framework by creating the two TVM backends to generate the 2PC version programs, which are instructed by the computations defined in the trained model. Figure 3 depicts the workflow of the TONIC framework. In particular, the two TVM backends generate the 2PC programs in the ObliVM language and in C++ with ABY function calls, respectively. In order to achieve high execution efficiency of the 2PC programs, the two backends should be aware of the characteristics of the target languages and need to perform the optimizations accordingly during the compilation, which are described in the following two subsections, respectively.

3.1 Characteristics of Oblivious Computing Languages

For the two oblivious computing languages, the language-level supports for writing an oblivious computing program are examined and the corresponding compilation strategies for these language supports are discussed.

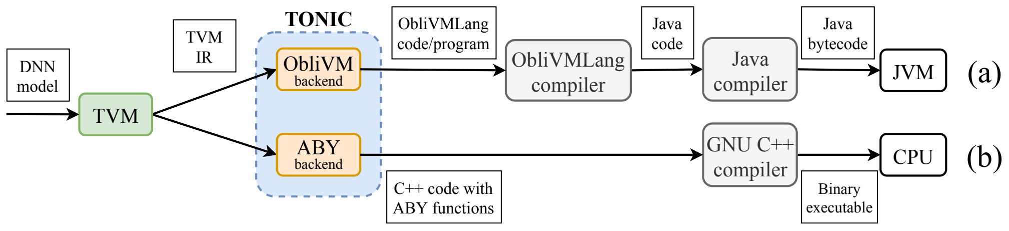

Figure 3: The workflow of the TONIC framework, which consists of two TVM backends for oblivious computing languages, ObliVM (a) and ABY (b), respectively.

3.1.1 Data Types. As ObliVM is a new domain-specific language for oblivious computing, its compiler currently supports the two primitive data types, `integer` and `float`, and the related mathematical operations (e.g., additions and multiplications) are available, which are sufficient to implement a variety of applications. In the TVM backend, `float` is adopted to keep the parameters of the given DL model. Internally, ObliVM relies on single-precision floating-point format to represent the ObliVM `float`, and based on our validation results, our compiled 2PC programs can achieve compatible accuracy against the original version.

On the other hand, ABY works with several bit schemes, such as 8, 16, 32, and 64 bits, for integers to support the multiple cryptographic protocols concurrently. Although it supports float operations for boolean protocol, what we want is to leverage the efficient arithmetic protocol, which is much faster than a boolean one. Therefore, we introduce the mechanism to scale up and down between floating-point numbers and fixed-point ones. The user image and the weights of the model are all converted into fixed-point before being inputted into ABY, and back to floating-point after the computation finishes. The 64-bit fixed-point data can be used for basic mathematical operations, so it may be used for DL inference computing. Nevertheless, while the 64-bit execution scheme provides better results (i.e., higher accuracy) for the model inference, it costs about 2x more time than the 32-bit fixed-point counterpart. Based on the validation results that we have done with ObliVM, TONIC-ABY uses the 32-bit fixed-point data representation for the floating-point data manipulations in the DL model, so as to balance the accuracy and the execution speed.

3.1.2 Data Structures. Typical CNNs contain multi-dimensional data for both user input data and model parameters, and it will save the TONIC backend development time if the oblivious computing languages support the multi-dimensional data structures (e.g., `vector`), and related computations (e.g., matrix-to-matrix or matrix-to-vector multiplications). Unfortunately, the multi-dimensional data storage and the related operations are not available in ObliVM, but only for ABY, which leverages the underlying C++ language support. Hence, in TONIC-ObliVM, the multi-dimensional data are converted into one-dimensional arrays in the generated 2PC programs. In addition, the supplement functions (e.g., the element-wise multiplication function) have been developed to emulate the original matrix/vector related operations.

Data variables used in ObliVM and ABY are either in plaintext (i.e., data contents are available to both parties, which is referred to as `public` data) or secretly shared (i.e., neither party sees the data contents, which is referred to as `secure` data). During the development of TONIC backends, we examine and determine the property for each type of data involved in the DL inference operations, and we assign the property (i.e., `public` or `secure` data) to each type of the data involved. For example, the user input data and the model parameters are stored as secure data to prevent leaking the data contents.

It is important to note that ObliVM implements the state-of-the-art oblivious structure, *Circuit ORAM* [33], to improve the privacy of both data contents and data access patterns, meaning that the data stored with the ORAM technology will not be revealed its content and the attackers are not able to guess the data contents by the access patterns. The access pattern hiding is done by performing the consistent access pattern, regardless of which data to be accessed; for instance, it could be done by walking through the entire memory locations whenever memory access needs to be performed. Based on our empirical results, the ORAM technology incurs a significant amount of runtime overheads to hide the data accesses, and hence, in TONIC-ObliVM, only the data contents are protected and leaving the data access patterns in plain sight. This design choice is consistent with our system model since the data access patterns are specified in the DL network architectures and it is easy to get the patterns by reverse engineering the machine codes of the 2PC executables.

3.1.3 Supplement Functions. The oblivious computing languages support some of the arithmetic operations for their primitive data types. For example, additions, multiplications, and divisions for the `float` data are supported by ObliVM. However, the basic operations commonly seen in a neural network model are often not supported by specialized languages. Taking the *softmax* operation as an example, it is widely adopted in the last layer of a CNN model to better rank the prediction results. It is not supported by either of the two languages as a built-in function, and it would be a challenge to implement the operation since *softmax* involves dividing and exponential operations incurring a high computational cost. They can be approximated by the Newton method or Taylor series in GC-based protocol (as adopted by ObliVM), but cannot be run in the sharing-based protocols (i.e., GMW-based protocols in ABY). According to the prior work [24], the softmax function can be ignored when performing the oblivious neural inference because it is the order-preserving operation and maximum and minimum data remain the same before and after applying the softmax operations.

3.2 Source-to-Source Translation

The considerations for the high-level code generation are described in this subsection. In particular, the following items are discussed: the execution model of the generated oblivious programs, the issues raised when generating codes for the different types of NN layers, the large code size problem for the generated code, and the huge memory footprint for keeping the input data during runtime.

3.2.1 *Execution Model.*

In a setting of 2PC with the GC-based protocol, there is a circuit *generator* and a circuit *evaluator*, and the data exchanges between them are performed via the *oblivious transfer (OT)*. At the initial stage of the GC-based execution scheme, they perform the following operations to facilitate the following execution: 1) the generator garbles the circuits; 2) the generator sends the garbled circuits and the labels of the circuit inputs to the evaluator; 3) the evaluator gets the labels of its inputs through OT; 4) the evaluator evaluates the garbled circuits. In step 3, the evaluator has to obtain the labels of its inputs from the generator through OT so that the evaluator is able to evaluate the garbled circuit sent by the generator later, and hence, the network traffic is determined by the input data size of the evaluator. Based on this observation, in order to improve the execution efficiency of oblivious neural inference, it is essential to determine which *roles* (i.e., circuit generator and evaluator) are played by the model provider and the user since the data sizes between a deep learning model and its input (i.e., user data) are often imbalanced. Therefore, it is always a good idea to let the one with the larger data size acting as the generator, and the communication volumes are reduced naturally. The setting of 2PC with GMW-based protocols does not have this kind of issue since data transmissions between the two parties are balanced. In summary, the above design is applied to TONIC-ObliVM and TONIC-ABY running with the GC mode, where the GC protocol in ABY is used to handle the non-linear layers.

3.2.2 *Layer-by-Layer Translation.*

TONIC adopts the layer-by-layer translation approach to generate the backend language codes for each type of the CNN layers, where there are mainly three types of layers found in most of the CNN models: 1) convolutional, 2) activation, and 3) fully-connected (FC) layers. As the convolutional and FC layers have to process a large number of structured data, TVM IR defines its own vector operations to handle the massive data manipulation, and these vector functions are mapped to the appropriate functions supported by the target hardware architecture by the TVM backend for computation accelerations. For example, the LLVM backend illustrated in Figure 1 may generate the AVX-based SIMD (Single Instruction Multiple Data) macros for these TVM vector operations to accelerate the matrix operations defined in CNN models on the Intel processors.

It is interesting to note that the *architectural SIMD support* in the garbled circuits is not available now, where the circuits can process multiple data with the same instruction concurrently to improve the processing speed, as the architectural SIMD support found in modern CPU/GPU. However, ABY does have its own SIMD support to reduce the generated circuit size, which is referred to as ABY-SIMD in this work. In particular, the ABY-SIMD uses the same circuit gate iteratively to process multiple data, instead of replicating multiple, identical circuit gates to process the multiple data, and the technique is able to reduce the circuit size, which further helps improve the execution efficiency of the 2PC programs because multiple garbled labels can be allocated in a single gate. In TONIC-ABY, the vector operations found in TVM IR are converted into the corresponding ABY-SIMD operations to improve the execution efficiency. On the other hand, ObliVM does not have a feature similar to ABY-SIMD, and in TONIC-ObliVM, the vector operations are converted into the array type operations.

Activation layer functions often contain non-linear operations, such as identifying the maximum and minimum numbers. Both ObliVM and ABY support such basic non-linear functions, e.g., `GT` returns `1` if the first operand on the left is greater than the second operand, otherwise, it returns `0`. As for those functions that are not supported, e.g., `max` and `min`, they are implemented by ourselves as the language extensions and these supplement functions invocations are generated when the corresponding non-linear operations are found during the layer-by-layer translation from TVM IR into the target oblivious computing language.

3.2.3 *Code Size Consideration.*

The Java virtual machine has the 64 KB code size limitation for the Java bytecode of a Java method[1]. During the early stage of the TONIC-ObliVM implementation, we found that some error occurs during the execution of the generated 2PC programs on the Java virtual machine. After further analysis, we find that the ObliVMLang compiler adopts the Static Single Assignment (SSA) form internally and generates the corresponding Java code from the SSA-enabled IR, which largely expands the code size of a generated Java method, since the SSA form will create new versions of the variable when there are multiple assignments to the very same variable, especially for the convolutional and fully-connected layers. To overcome this problem, the TONIC-ObliVM backend generates the ObliVM program in the SSA form, which helps prevent expanding the code size. On the other hand, we do not find a similar issue in ABY, which adopts the C++ library support for 2PC and relies on the third-party compiler toolchains to compile the ABY program.

3.2.4 *Memory Footprint Reduction.*

Conventionally, the inputs are fed into the 2PC programs during the initialization of both programs before entering the *main* entry points. Nevertheless, since ObliVM requires the data to be stored in the binary format (i.e., using the binary representation of floating-point numbers and saving as the binary strings), the size of the memory used to keep the input data is grown proportionally with the size of the input. This incurs the problem when running with a large CNN model. Taking VGG16 as an example, it requires about 500 MB to store its model parameters in the file system, and each 2PC program will need 12 GB of memory to proceed with the execution.

As most of the model parameters are local to a certain NN layer, TONIC-ObliVM uses the dynamic parameter loading technique to feed only the required parameters to a NN layer as needed at the beginning of the execution of the layer, which effectively reduces the peak memory footprint. In addition, the binary representation of the input data consumes a large amount of space so the data are saved in the floating-point format to save the space. Besides,

[1]The `code_length` item of the `Code` attributes: https://docs.oracle.com/javase/specs/jvms/se11/html/jvms-4.html#jvms-4.7.3

since the secure data content requires relatively larger space than its public version, TONIC-ObliVM saves the dynamically loaded parameters of a layer in the public form and converts the public data into the corresponding secure version only when the data are required for the secure computations. The above dynamic parameter loading and conversing techniques conform to the security model defined in Section 2.4 because the model parameters in the plaintext are only available to the model provider and the user is allowed to see the encrypted version of the parameters during the computation if necessary.

Figure 4 illustrates the concept of the dynamic parameter loading for each layer as needed, where the model provider acts as the *generator* and the user is the *evaluator*, as described in Section 3.2.1. Note that the generator adopts the dynamic parameter loading and secure data conversion schemes, and the evaluator uses the default mechanism provided by ObliVM to feed the input data (i.e., reading the input image at once during the program initialization stage and saving the data as secure contents by default). Note that while ABY does not have the memory footprint issue as the ObliVM does, TONIC-ABY uses the dynamic parameter loading technique for each NN layer in order to reduce the peak memory footprint to facilitate the execution of 2PC programs.

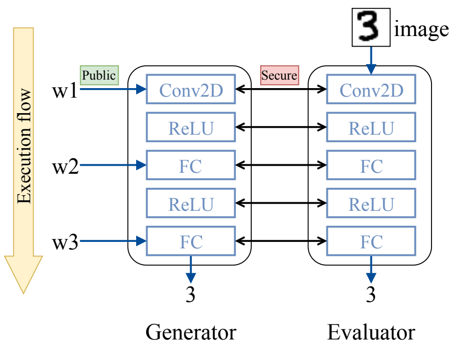

Figure 4: Illustration of the dynamic parameter loading technique adopted by TONIC.

4 EXPERIMENTAL RESULTS

The capabilities of the proposed framework are demonstrated in this section. Thanks to TVM, which accepts various popular deep learning frameworks, such as Keras, TensorFlow, and PyTorch, TONIC can handle the models trained by these TVM-supported frameworks. We show that TONIC is able to convert a variety of the trained CNN models into the corresponding 2PC versions without any manual conversion efforts. Specifically, we use Keras [12] to build and train the well-known models, which are introduced in Section 4.1, for the classification problems with the datasets, MNIST and ImageNet, since we are not able to find the pre-trained models online. The trained models are imported into TVM via its Keras frontend, and TONIC backends are responsible for generating the 2PC version codes. The delivered performance of the 2PC programs between TONIC and the prior work is presented in Section 4.2. Section 4.3 validates the computation results of the 2PC programs generated by TONIC. The memory footprints consumed by the generated oblivious neural inference codes are evaluated in Section 4.4.

Most of the experiments are performed on the two machines in the LAN setting. One machine is equipped with Intel Core i7-10700K CPU with 32 GB memory, and the other has Intel Core i7-7700K with 16 GB memory, where Ubuntu 18.04 is run on both machines that are connected with each other via the Gigabit Ethernet.

4.1 Neural Network Workloads

Based on the settings in [18, 24, 25, 29], we create the four CNN models for the MNIST dataset. Furthermore, we also created the large model, MobileNetV2, for handling the ImageNet dataset. Their architectures are listed below. The training accuracy of the built CNN models is given in Table 1.

- **BM1.** FC-Square-FC-Square-FC, from [25].
- **BM2-1.** Conv2d-Square-FC-Square-FC, from [18].
- **BM2-2.** Conv2d-ReLU-FC-ReLU-FC, from [29].
- **BM3.** Conv2D-ReLU-Maxpool-Conv2D-ReLU-Maxpool-FC-ReLU-FC, from [24].
- **MobileNetV2 (0.35-224).** 52 Convolutions, 35 ReLU6s, 1 Average pooling, 1 FC, from [30].

Table 1: The accuracy of the trained models.

Dataset	Arch.	Trained Top-1 Acc. (%)
MNIST	BM1	97.63
	BM2-1	98.80
	BM2-2	98.72
	BM3	99.18
ImageNet	MobileNetV2	60.3 (Top-5: 82.9)

4.2 Execution Time

Two sets of performance results are presented according to the used datasets, MNIST and ImageNet, which are presented in Sections 4.2.1 and 4.2.2, respectively. Especially, the delivered performance achieved by TONIC and the prior work is compared to demonstrate the capability of TONIC.

4.2.1 CNNs for MNIST. The runtime of the 2PC programs generated by TONIC and the prior work is listed in Table 2. It is obvious that the TONIC-ObliVM version has the longest runtime among others since ObliVM uses hash functions and the on-the-fly circuit generation and execution that incurs more frequent communications between the two machines when evaluating every single gate. In addition, ObliVM is based on the Java virtual machine, which would introduce another execution overhead. Based on our results, while ObliVM provides a higher-level programming interface to alleviate the programming overhead and uses the on-the-fly execution to mitigate the large circuit size issue, they are achieved at the cost of higher running time.

TONIC-ABY, on the other hand, achieves comparable performance, compared with the other approaches. In particular, the programs generated by TONIC-ABY have moderate runtime, which is higher than Gazelle and XONN. The major advantage of these two works is that they produce highly optimized models by manually adjusting circuit designs and by re-training the models optimizing for binarized neural networks, respectively. This is the reason why these two works are significantly faster than the others. Nevertheless, it will require a remarkable development time when handling a different model and it would be hard for these two works to generate the 2PC version of a larger model due to the same reason which makes them faster. More details about the two works are given in Section 5.3.

By further decomposing the runtime into the *offline* and *online* parts, it is shown that the online time is more close to the leading group. For example, TONIC-ABY is only two to four times slower than Gazelle, but it is able to generate the trained CNN models, which is not applicable for Gazelle. In fact, the prior work listed in Table 2 does not support the automatic model conversion to the 2PC version, and TONIC-ABY is faster than some of them. Based on the online time, we believe that TONIC-ABY is with sufficient speed and is able to be put to real use. For example, the secure CNN inference service is a promising candidate application since the circuits can be pre-built and loaded to eliminate the setup (offline) time.

Table 2: Comparison of TONIC with the state-of-the-art for the MNIST neural inference in LAN setting. Runtimes are in seconds.

Arch.	Framework	Offline	Online	Runtime
BM1	SecureML	4.7	0.18	4.88
	MiniONN	0.9	0.14	1.04
	EzPC [a]	—	—	0.7
	Gazelle	0	0.03	0.03
	XONN [a]	—	—	0.13
	TONIC-ObliVM [b]	0	955.5	955.5
	TONIC-ABY	0.776	0.123	0.899
BM2-1	CryptoNets	—	—	297.5
	MiniONN	0.88	0.4	1.28
	HyCC	0.683	0.134	0.817
	EzPC [a]	—	—	0.6
	Gazelle	0	0.03	0.03
	TONIC-ObliVM [b]	0	989.6	989.6
	TONIC-ABY	0.704	0.095	0.799
BM2-2	DeepSecure	—	—	9.67
	HyCC	0.784	0.163	0.947
	Gazelle	0.15	0.05	0.20
	XONN [a]	—	—	0.13
	TONIC-ObliVM [b]	0	907	907
	TONIC-ABY	0.729	0.102	0.831
BM3	MiniONN	3.580	5.740	9.32
	HyCC	1.825	1.621	3.446
	EzPC [a]	—	—	5.1
	Gazelle	0.481	0.33	1.16
	XONN [a]	—	—	0.15
	TONIC-ObliVM [b]	0	4844	4844
	TONIC-ABY	4.929	1.257	6.186

[a] The decomposition of the runtime cost into offline and online is not reported by the authors.
[b] We run all ObliVM experiments in the localhost setting and the offline is inaccessible because of the on-the-fly technique.

4.2.2 MobileNetV2 for ImageNet. Table 3 lists the performance of the 2PC programs generated by TONIC-ABY and nGraph-HE2, the state-of-the-art for automatically compiling a CNN model for secure neural inference with homomorphic encryption (HE). The results show that TONIC is 2.3× faster than nGraph-HE2 when considering the latency for handling a single input image data. It is important to note that nGraph-HE2 is designed for high throughput systems, where it has a similar latency for handling 4,096 images in one batch with 56 threads. On the other hand, our approach requires no more than two threads for handling a single secure inference request at a time, which is more suitable for the medical application mentioned in Section 1, where each user can protect the privacy of its own data and it does not require the packing of other user data together for oblivious secure inference at the risk of leaking the data during the data packing. While TONIC-ABY and nGraph-HE2 are capable of generating the 2PC version of the deep model, MobileNetV2, they are different in the adopted cryptographic protocols, detailed comparison of the two works is in Section 5.4.

Table 3: Performance delivered by TONIC with nGraph-HE2 for MobileNetV2. Runtimes are in seconds.

Framework	Runtime					
	Localhost			LAN		
	Offline	Online	Total	Offline	Online	Total
nGraph-HE2 [a]	—	—	529	—	—	1,559
TONIC-ABY	162	62	224	502	123	625

[a] Separation of runtime cost into offline and online is not reported by the authors.

4.3 Validation

To validate the computation results of the 2PC version programs against those produced by the original models. For the 2PC version, the *verify* mode offered by ObliVM is turned on to obtain the computational outputs after the inference operations performed on each layer. As for the original model, we have revised the C++ backend from TVM and run the model inference in the TVM debug mode to obtain the outcomes for each model layer. The layer-by-layer outputs comparison between the 2PC and the original versions uses mean absolute error (MAE) to quantify the difference between them.

Figure 5 shows the MAEs between the two versions of the BM2-2 model inference on the MNIST dataset. Overall, the accumulated MAE, observed after performing the `softmax` operations at the last layer of the model, is at the order of magnitude of −4 (i.e., 1.12e−4). Based on our preliminary analyses, the error would be originated from the floating-point numbers system adopted by ObliVM, where the IEEE 754 like system is adopted for computing the 32-bit floating-point data, and the C++ backend generates the 32-bit floating-point numbers run by C++ language conforming IEEE 754 standard. It can be seen that the error is increased with the involved multiplication operations. For instance, there is a steep rise after the `Conv2d` (convolutional) operations performed and another is at the dense (fully-connected) layer. It is interesting to note that the non-linear layer, such as ReLU, actually helps reduce the error since the original and 2PC versions have similar numerical values and the non-linear operations convert these similar values into the same output values.

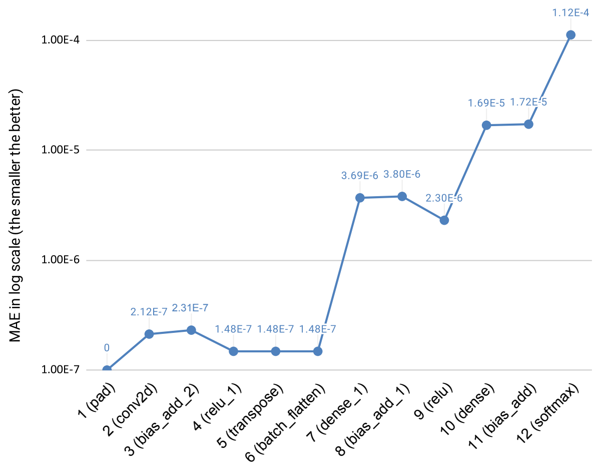

Figure 5: Average MAE for each layer of ObliVM backend of BM2-2 on the MNIST dataset.

4.4 Memory Footprint

Table 4 lists the peak memory required during the oblivious neural inference for the BM2-2 model with MNIST dataset. We find that the on-the-fly technique adopted by ObliVM reduces lots of memory usage so that the Java programs can run under the small memory footprint. On the other hand, ABY has to construct the entire circuits first before its execution, so the memory consumption is higher than that required by ObliVM.

Table 4: Peak memory footprint of the 2PC for BM2-2.

Framework	Peak memory footprint (MB)	
	Generator	Evaluator
TONIC-ObliVM	16	16
TONIC-ABY	92	83

5 RELATED WORK

This section introduces the software efforts for the general-purpose oblivious computations. Furthermore, on top of the software infrastructures, the privacy-preserving inference techniques via the handcraft and the automatic compilation approaches are described. As this work belongs to the automatic compilation category, we discuss the major differences between TONIC and the prior work.

5.1 General-purpose Oblivious Computing Software

Many research efforts have been done to provide the software platforms facilitating the oblivious computations. *JustGarble* [2] provides a C library for garbling and evaluating circuits based on the fixed-key AES, where the AES computations are accelerated by the CPU, e.g., Intel AES-NI. Similarly, ABY [16] offers C++ libraries to support the mixed-protocol 2PC and to efficiently convert among the three protocols, arithmetic sharing of GMW, boolean sharing of GMW, and Yao's GC. It is shown that *oblivious transfer extension (OT extension)* [20] is more efficient than HE (homomorphic encryption) schemes (*DGK* [14] and *Paillier* [26]) for precomputing multiplication triples of arithmetic sharing [16]. There are also studies of building the programming languages for oblivious computations. Usually, these studies are either proposing domain-specific languages or using the lower-level languages, e.g., C language. *ObliVM* [23] and *Obliv-C* [35] are such examples, where the two general-purpose frameworks take the program written in their supported languages as inputs and generate the executables by leveraging other 2PC software; for example, ObliVM uses its internal library *ObliVMGC* and Obliv-C adopts JustGarble to perform the GC protocol. *EzPC* [8] proposes the domain-specific language for the 2PC, which leverages the ABY support for running between the boolean sharing and arithmetic sharing protocols. *HyCC* [7], on the other hand, converts the user given C code to the different versions of circuits (i.e., using different combinations of the 2PC protocols) by leveraging the ABY library and HyCC attempts to select the best protocol for 2PC execution based on the C code structures.

5.2 Handcrafted SMPC for Specific NN Models

To shorten the execution time for privacy-preserving inferences, works requiring human intervention have been done by manually crafting implementations of NN models with the support of existing low-level toolkits. *CryptoNets* [18] is the first work to approach the privacy-preserving inference, and it uses *YASHE′* [5] scheme and is implemented by SEAL [31]. *DeepSecure* [29] is based on the GC protocol to achieve classification and has the mechanism to reduce the computations through preprocessing. In addition to a single cryptographic protocol, the hybrid-protocol based schemes are also developed. *MiniONN* [24] mixes *YASHE* implemented by SEAL and ABY's GC and arithmetic sharing. *Gazelle* [21], the state-of-the-art in this category, combines GC and *packed additive homomorphic encryption (PAHE)* scheme, which leverages the PALISADE [28] library and benefits from the SIMD operations by packing multiple plaintexts into a single ciphertext. Gazelle calculates all linear computations (e.g., operations in convolutional and FC layers) within the PAHE scheme and transfers to GC for the non-linear layers (e.g., ReLU and MaxPooling). *SecureML* [25] leverages ABY to handle the conversion between GC and arithmetic sharing for the 2PC of machine learning applications, including linear regression and logistic regression, as well as neural network training and inference.

5.3 Automatic Conversion of NN Models for SMPC

Some works have been developed to take a NN model, which is a trained model or is described via the high-level descriptions, such as Python using *Keras* [12], as the input and to convert the NN model into the 2PC programs for privacy-preserving inference. *CHET* [15] is known to be the pioneer to introduce the compiler and runtime for performing such a conversion task and using the FHE (fully homomorphic encryption) scheme for the 2PC. *nGraph-HE* [4] and *nGraph-HE2* [3] leverage the graph compiler (i.e., *Intel nGraph* [13]) to convert a given DNN model to the 2PC programs running with the HE-based protocol. As the nGraph compiler is similar to TVM, it is able to support DNN models imported from a variety of software frameworks, such as *TensorFlow* [1] and *MXNet* [9]. Both nGraph-HE and nGraph-HE2 are the backends for nGraph compiler for generating the 2PC programs with the supports from BFV [6, 17] and CKKS [11], both powered by SEAL. nGraph-HE2 optimizes CKKS to achieve 2x higher throughput than nGraph-HE. In addition, the client-aided method is adopted to decrease the multiplicative depth, which further reduces the execution time. Nevertheless, the client-aided scheme requires the transfer of the intermediate outputs from the server to the client and hence, have the potential of leaking the data contents, which means the privacy of the model parameters is compromised since the client can reveal the raw information from the intermediate data. *XONN* [27] provides the frontend to parse the model described by Keras, and it requires to train the given network architecture using the Binary Neural Networks (BNNs) in order to find a proper configuration so that the resulting BNN-based model performs well. While the BNN-based approach helps accelerate the computation of GC protocol, the method is not suitable for the applications described in Section 1, which relies on the incremental trained DL model and the re-training of the built DL model is time-consuming and requires the entire training datasets in history.

5.4 Discussion

Our proposed framework is similar to nGraph-HE and nGraph-HE2 in that TONIC can operate on the trained deep learning models, which are accepted by the graph compiler, TVM. The major difference is that TONIC runs on top of the SMPC-based protocols, whereas the nGraph-based approaches adopt the HE-based schemes for the privacy-preserving inference. As a result, TONIC and nGraph-based approaches have different security models [4]. Furthermore, TONIC-ABY has relatively shorter latency for handling the oblivious neural inference of a single input data for MobileNetV2, as shown in Section 4, and does not have the potential security risk as nGraph-HE2 does. Therefore, we believe that TONIC is applicable for real-world applications, such as the medical task mentioned in Section 1.

6 CONCLUSION

In this work, we propose the TONIC framework for compiling trained CNN models into the corresponding 2PC programs for oblivious neural inference. We describe the issues faced by the oblivious computing software, ObliVM and ABY, and our devised compilation strategies. Our results show that our two TONIC backends are able to convert a variety of trained CNN models. Especially, the TONIC-ABY backend is able to convert the deep neural network, MobileNetV2, which shows our approach is scalable and the execution time of the 2PC-based MobileNetV2 is faster than the inference time for the 2PC MobileNetV2 generated by the prior work, nGraph-HE and nGraph-HE2. We believe that our work is applicable to the oblivious neural inference for real-world applications.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/. Software available from tensorflow.org.

[2] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. 2013. Efficient Garbling from a Fixed-Key Blockcipher. In *IEEE Symposium on Security and Privacy*. 478–492.

[3] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 45–56.

[4] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *ACM International Conference on Computing Frontiers*. 3–13.

[5] Joppe W. Bos, Kristin E. Lauter, Jake Loftus, and Michael Naehrig. 2013. Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme. In *IMA International Conference on Cryptography and Coding*. 45–64.

[6] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Annual International Cryptology Conference*. 868–886.

[7] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *ACM Conference on Computer and Communications Security*. 847–861.

[8] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. 2019. EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning. In *IEEE European Symposium on Security and Privacy*. 496–511.

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *Computing Research Repository* abs/1512.01274 (2015).

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation*. 578–594.

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, Vol. 10624. 409–437.

[12] François Chollet et al. 2015. Keras. https://keras.io.

[13] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *Computing Research Repository* abs/1801.08058 (2018).

[14] Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. 2008. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography* 1 (2008), 22–31.

[15] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin E. Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 142–156.

[16] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Network*

and Distributed System Security Symposium.

[17] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2012 (2012), 144.

[18] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *International Conference on Machine Learning*. 201–210.

[19] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *ACM Symposium on Theory of Computing*. 218–229.

[20] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In *Annual International Cryptology Conference*. 145–161.

[21] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *USENIX Security Symposium*. 1651–1669.

[22] Joe Kilian. 1988. Founding Cryptography on Oblivious Transfer. In *ACM Symposium on Theory of Computing*. 20–31.

[23] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *IEEE Symposium on Security and Privacy*. 359–376.

[24] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN Transformations. In *ACM Conference on Computer and Communications Security*. 619–631.

[25] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE Symposium on Security and Privacy*. 19–38.

[26] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*. 223–238.

[27] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. 2019. XONN: XNOR-based Oblivious Deep Neural Network Inference. In *USENIX Security Symposium*. 1501–1518.

[28] Kurt Rohloff and Yuriy Polyakov. 2017. The PALISADE Lattice Cryptography Library. https://git.njit.edu/palisade/PALISADE. 1.0 edition.

[29] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. 2018. Deepsecure: scalable provably-secure deep learning. In *Design Automation Conference*. 1–6.

[30] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4510–4520.

[31] SEAL 2019. Microsoft SEAL (release 3.4). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[32] Adi Shamir, Ronald L Rivest, and Leonard M Adleman. 1979. *Mental Poker*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.

[33] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. 2015. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM Conference on Computer and Communications Security*. 850–861.

[34] Andrew Chi-Chih Yao. 1982. Protocols for secure computations. In *IEEE Symposium on Foundations of Computer Science*. 160–164.

[35] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptology ePrint Archive* 2015 (2015), 1153.