Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications

M. Sadegh Riazi
UC San Diego
mriazi@eng.ucsd.edu

Christian Weinert
TU Darmstadt, Germany
christian.weinert@crisp-da.de

Oleksandr Tkachenko
TU Darmstadt, Germany
oleksandr.tkachenko@crisp-da.de

Ebrahim M. Songhori
UC San Diego
e.songhori@gmail.com

Thomas Schneider
TU Darmstadt, Germany
thomas.schneider@crisp-da.de

Farinaz Koushanfar
UC San Diego
fkoushanfar@eng.ucsd.edu

ABSTRACT

We present Chameleon, a novel hybrid (mixed-protocol) framework for secure function evaluation (SFE) which enables two parties to jointly compute a function without disclosing their private inputs. Chameleon combines the best aspects of generic SFE protocols with the ones that are based upon additive secret sharing. In particular, the framework performs linear operations in the ring $\mathbb{Z}_{2^l}$ using additively secret shared values and nonlinear operations using Yao's Garbled Circuits or the Goldreich-Micali-Wigderson protocol. Chameleon departs from the common assumption of additive or linear secret sharing models where three or more parties need to communicate in the online phase: the framework allows two parties with private inputs to communicate in the online phase under the assumption of a third node generating correlated randomness in an offline phase. Almost all of the heavy cryptographic operations are precomputed in an offline phase which substantially reduces the communication overhead. Chameleon is both scalable and significantly more efficient than the ABY framework (NDSS'15) it is based on. Our framework supports signed fixed-point numbers. In particular, Chameleon's vector dot product of signed fixed-point numbers improves the efficiency of mining and classification of encrypted data for algorithms based upon heavy matrix multiplications. Our evaluation of Chameleon on a 5 layer convolutional deep neural network shows 133x and 4.2x faster executions than Microsoft CryptoNets (ICML'16) and MiniONN (CCS'17), respectively.

KEYWORDS

Secure Computation; Garbled Circuits; Secret Sharing; Deep Neural Networks; Machine Learning

ACM Reference Format:
M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3196494.3196522

1 INTRODUCTION

Secure Function Evaluation (SFE) is one of the great achievements of modern cryptography. It allows two or more parties to evaluate a function on their inputs without disclosing the inputs to each other; that is, all inputs are kept private by the respective owners. In fact, SFE emulates a *trusted* third party which collects inputs from different parties and returns the result of the function to all (or a specific set of) parties. SFE has many applications in privacy-preserving biometric authentication [17, 78], secure auctions [38], secure search [76], privacy-preserving machine learning [36], and data mining [60, 70]. The two most prominent SFE protocols are Yao's Garbled Circuits (GC) [85] and the Goldreich-Micali-Wigderson (GMW) protocol [41].

In theory, any function that can be represented as a Boolean circuit can be evaluated securely using the GC or GMW protocol. However, GC and GMW can often be too slow and hence are of limited practical value because they require several symmetric key operations for each gate in the circuit. During the past three decades, the great effort of the secure computation community has decreased the overhead of SFE protocols by several orders of magnitude. The innovations and optimizations span the full range from protocol-level to algorithm-level to engineering-level. As a result, several frameworks have been designed with the goal of efficiently realizing one (or multiple) SFE protocols. They vary by the offline/online runtime, the number of computing nodes (two-party or multi-party), offline/online communication, the set of supported instructions, and the programming language which describes the functionality. These frameworks accept the description of the function as either (i) their own customized languages [65, 68], (ii) high-level languages such as `C/C++` [45] or `Java` [47, 61], or (iii) Hardware Description Languages (HDLs) [33, 83].

A number of SFE compilers have been designed for translating a program written in a high level language to low-level code [42, 65, 68]. The low-level code is supported by other SFE frameworks which serve as a backbone for executing the cryptographic protocols. In addition to generic SFE protocols, additive/linear *secret sharing* enables secure computation of linear operations such as multiplication, addition, and subtraction. In general, each framework introduces a set of trade-offs. The frameworks based on secret-sharing require three (or more) computing nodes which operate on distributed shares of variables in parallel and require multiple rounds of communication between nodes to compute an operation on shares of two secret values.

One of the most efficient secure computation frameworks is Sharemind [18] which is based on Additive Secret Sharing (A-SS) over the specific ring $\mathbb{Z}_{2^{32}}$. All operations are performed by three computing nodes. Sharemind is secure against honest-but-curious (semi-honest) nodes which are assumed to follow the protocol but they cannot infer any information about the input and intermediate results as long as the majority of nodes are not corrupted. We consider the same adversary model in this paper. Securely computing each operation in Sharemind needs multiple communication rounds between all three nodes which makes the framework relatively slow in the Internet setting. Computation based on additive shares in the ring $\mathbb{Z}_{2^l}$ enables very efficient and fast linear operations such as Multiplication (MULT), Addition (ADD), and Subtraction (SUB). However, operations such as Comparison (CMP) and Equality test (EQ) are not as efficient and *non-linear* operations cannot easily be realized in the ring $\mathbb{Z}_{2^l}$.

We introduce Chameleon, a fast, modular, and hybrid (mixed-protocol) secure two-party computation framework that utilizes GC, GMW, and additive secret sharing protocols and achieves unprecedented performance both in terms of run-time and communication between parties. The analogy comes from the fact that similar to a chameleon that changes its color to match the color of the environment, our framework allows changing the executing SFE protocol based on the run-time operation. The main design goal behind Chameleon is to create a framework that combines the advantages of the previous secure computation methodologies.

The idea of a mixed-protocol solution was first introduced in [21] which combines GC with Homomorphic Encryption (HE). HE enables to perform MULT and ADD operations on encrypted values without actually knowing the unencrypted data. The TASTY framework [42] enables automatic generation of protocols based on GC and HE. However, due to the high computational cost of HE and costly conversion between HE and GC, they achieve only a marginal improvement compared to the single protocol execution model [51].

Our framework Chameleon is based on ABY [35] which implements a hybrid of additive SS, GMW, and GC for efficient realization of SFE. However, we overcome two major limitations, thereby improving efficiency, scalability, and practicality: The ABY model relies on oblivious transfers for precomputing arithmetic triples which we replace by more efficient protocols using a Semi-honest Third Party (STP). The STP can be a separate computing node or it can be implemented based on a smartcard [34] or Intel Software Guard Extensions (SGX) [7]. Therefore, the online phase of Chameleon only involves two parties that have private inputs. Additionally, we extend ABY to handle signed fixed-point numbers which is needed in many deep learning applications, but not provided by ABY and other state-of-the-art secure computation frameworks such as TASTY.

Chameleon supports 16, 32, and 64 bit signed fixed-point numbers. The number of bits assigned to the fraction and integral part can also be tuned according to the application. The input programs to Chameleon can be described in the high-level language `C++`. The framework itself is also written in `C++` which delivers fast execution. In addition to a rich library of pre-defined functions, the user can simply add any function description as a Boolean circuit or a `C/C++` program to our framework and use them seamlessly.

Machine Learning on Private Data Using Chameleon. Chameleon's efficiency helps us to address a major problem in contemporary secure machine learning on private data. Matrix multiplication (or equivalently, vector dot product computation) is one of the most frequent and essential building blocks for many machine learning algorithms and applications. Therefore, in addition to scalability and efficiency described earlier, we design an efficient secure vector dot product protocol based on the Du-Atallah multiplication protocol [37] that has very fast execution and low communication between the two parties. We address secure Deep Learning (DL) which is a sophisticated task with increasing attraction. We also provide privacy-preserving classification based on Support Vector Machines (SVMs).

The fact that many pioneering technology companies have started to provide Machine Learning as a Service (MLaaS[1,2,3]) proves the importance of DL. Deep and Convolutional Neural Networks (DNNs/CNNs) have attracted many machine learning practitioners due to their capabilities and high classification accuracy. In MLaaS, clients provide their inputs to the cloud servers and receive the corresponding results. However, the privacy of clients' data is an important driving factor. To that end, Microsoft Research has announced CryptoNets [36]. CryptoNets is an HE-based methodology that allows secure evaluation (inference) of encrypted queries over *already trained* neural networks on cloud servers: queries from the clients can be classified securely by the trained neural network model on a cloud server without inferring any information about the query or the result. In §6.1, we show how Chameleon improves over CryptoNets as well as other previous works. In addition, we evaluate Chameleon for privacy-preserving classification based on Support Vector Machines in App. B.

Our Contributions. Our main contributions are as follows:

- We introduce Chameleon, a novel mixed SFE framework based on ABY [35] which brings benefits in terms of efficiency, scalability, and practicality by integrating signed fixed-point arithmetic, STP-based protocols for precomputing OTs and generating arithmetic and Boolean multiplication triples, and an optimized STP-based vector dot product protocol for vector/matrix multiplications.
- We provide detailed performance evaluation results of Chameleon compared to state-of-the-art frameworks. Compared to ABY, Chameleon requires up to 321× and 256× less communication for generating arithmetic and Boolean multiplication triples, respectively.
- We present a proof-of-concept implementation and experimental results on deep and convolutional neural networks. Comparing to the state-of-the-art Microsoft CryptoNets [36], we achieve a 133x performance improvement. Comparing to the recent work of [62], we achieve a 4.2x performance improvement using a comparable configuration.

[1]Amazon AWS AI (https://aws.amazon.com/amazon-ai/)
[2]Google Cloud Machine Learning Engine (https://cloud.google.com/ml-engine/)
[3]Microsoft Azure Machine Learning Services (https://azure.microsoft.com/services/machine-learning-services/)

2 PRELIMINARIES

In the following, we provide a concise overview of the basic protocols and concepts that are used in the paper. Intermediate values are kept as secret shares of different types. We denote a share of value x, in type T, and held by party i as $\langle x\rangle_i^T$.

2.1 Oblivious Transfer Protocol

Oblivious Transfer (OT) is a building block for secure computation protocols. The OT protocol allows a receiving party $\mathcal{R}$ to obliviously select and receive a message from a set of messages that belong to a sending party $\mathcal{S}$, i.e., without letting $\mathcal{S}$ know which message was selected. In 1-out-of-2 OT, $\mathcal{S}$ has two l-bit messages x_0, x_1 and $\mathcal{R}$ has a bit b indicating the index of the desired message. After performing the protocol, $\mathcal{R}$ obtains x_b without learning anything about x_{1-b} and $\mathcal{S}$ learns no information about b. We denote n parallel 1-out-of-2 OTs on l-bit messages as OT_l^n.

The OT protocol requires costly public-key cryptography that significantly degrades the performance of secure computation. A number of methods have been proposed to perform a large number of OTs using only a few public-key encryptions together with less costly symmetric key cryptography in a constant number of communication rounds [6, 13, 48]. Although the OT extension methods significantly reduce the cost compared to that of the original OT, the cost is still prohibitively large for complex secure computation that relies heavily on OT. However, with the presence of a semi-trusted third party, the parties can perform OT protocols with very low cryptographic computation cost as explained in §5.5.

2.2 Garbled Circuit Protocol

One of the most efficient solutions for generic secure two-party computation is Yao's Garbled Circuit (GC) protocol [85] that requires only a constant number of communication rounds. In the GC protocol, two parties, Alice and Bob, wish to compute a function $f(a,b)$ where a is Alice's private input and b is Bob's. The function $f(.,.)$ has to be represented as a Boolean circuit consisting of two-input gates, e.g., AND and XOR. For each wire w in the circuit, Alice generates and assigns two random k-bit strings, called *labels*, X_w^0 and X_w^1 representing 0 and 1 Boolean values where k is a security parameter, usually set to $k = 128$ [14]. Next, she encrypts the output labels of a gate using the two corresponding input labels as the encryption keys and creates a four-entry table called *garbled table* for each gate. The garbled table's rows are shuffled according to the point-and-permute technique [69] where the four rows are permuted by using the Least Significant Bit (LSB) of the input labels as the permutation bits. Alice sends the garbled tables of all the gates in the circuit to Bob along with the labels corresponding to her input a. Bob also obliviously receives the labels for his inputs from Alice through OT. He then decrypts the garbled tables one by one to obtain the output labels of the circuit's output wires. Alice on the other hand has the mapping of the output labels to 0 and 1 Boolean values. They can learn the output of the function by sharing this information.

2.3 GMW Protocol

The Goldreich-Micali-Wigderson (GMW) protocol is an interactive secure multi-party computation protocol [40, 41]. In the two-party GMW protocol, Alice and Bob compute $f(a,b)$ using secret-shared values, where a is Alice's private input and b is Bob's. Similar to the GC protocol, the function $f(.,.)$ has to be represented as a Boolean circuit. In GMW, the Boolean value of a wire in the circuit is shared between the parties: Alice has $\langle v\rangle_0^B$, Bob has $\langle v\rangle_1^B$, and the actual Boolean value is $v = \langle v\rangle_0^B \oplus \langle v\rangle_1^B$. Since the XOR operation is associative, the XOR gates in the circuit can be evaluated locally and without any communication between the parties. The secure evaluation of AND gates requires interaction and communication between the parties. The communication for the AND gates on the same level of the circuit can be done in parallel. Suppose an AND gate $x \wedge y = z$ (where $\wedge$ is the AND operation) where Alice has shares $\langle x\rangle_0^B$ and $\langle y\rangle_0^B$, Bob has shares $\langle x\rangle_1^B$ and $\langle y\rangle_1^B$, and they wish to obtain shares $\langle z\rangle_0^B$ and $\langle z\rangle_1^B$, respectively.

As shown in [35], the most efficient method for evaluating AND gates in the GMW protocol is based on Beaver's multiplication triples [11]: Multiplication triples are random shared-secrets a, b, and c such that $\langle c\rangle_0^B \oplus \langle c\rangle_1^B = (\langle a\rangle_0^B \oplus \langle a\rangle_1^B) \wedge (\langle b\rangle_0^B \oplus \langle b\rangle_1^B)$. The triples can be generated offline using OTs (cf. [80]) or by a semi-trusted third party (cf. §5.4). During the online phase, Alice and Bob use the triples to mask and exchange their inputs of the AND gate: $\langle d\rangle_i^B = \langle x\rangle_i^B \oplus \langle a\rangle_i^B$ and $\langle e\rangle_i^B = \langle y\rangle_i^B \oplus \langle b\rangle_i^B$. After that, both can reconstruct $d = \langle d\rangle_0^B \oplus \langle d\rangle_1^B$ and $e = \langle e\rangle_0^B \oplus \langle e\rangle_1^B$. This way, the output shares can be computed as $\langle z\rangle_0^B = (d \wedge e) \oplus (\langle b\rangle_0^B \wedge d) \oplus (\langle a\rangle_0^B \wedge e) \oplus \langle c\rangle_0^B$ and $\langle z\rangle_1^B = (\langle b\rangle_1^B \wedge d) \oplus (\langle a\rangle_1^B \wedge e) \oplus \langle c\rangle_1^B$.

2.4 Additive Secret Sharing

In this protocol, a value is shared between two parties such that the addition of two secrets yields the true value. All operations are performed in the ring $\mathbb{Z}_{2^l}$ (integers modulo 2^l) where each number is represented as an l-bit integer. A ring is a set of numbers which is closed under addition and multiplication.

In order to additively share a secret x, a random number within the ring is selected, $r \in_R \mathbb{Z}_{2^l}$, and two shares are created as $\langle x\rangle_0^A = r$ and $\langle x\rangle_1^A = x - r \bmod 2^l$. A party that wants to share a secret sends one of the shares to the other party. To reconstruct a secret, one only needs to add the two shares $x = \langle x\rangle_0^A + \langle x\rangle_1^A \bmod 2^l$.

Addition, subtraction, and multiplication by a public constant value η ($z = x \circ \eta$) can be done locally by the two parties without any communication: party i computes the share of the result as $\langle z\rangle_i^A = \langle x\rangle_i^A \circ \eta \bmod 2^l$, where $\circ$ denotes any of the aforementioned three operations. Adding/subtracting two secrets ($z = x \pm y$) also does not require any communication and can be realized as $\langle z\rangle_i^A = \langle x\rangle_i^A \pm \langle y\rangle_i^A \bmod 2^l$. Multiplying two secrets, however, requires one round of communication. Furthermore, the two parties need to have shares of precomputed Multiplication Triples (MTs). MTs refer to a set of three shared numbers such that $c = a \times b$. In the offline phase, party i receives $\langle a\rangle_i^A$, $\langle b\rangle_i^A$, and $\langle c\rangle_i^A$ (cf. §5.4). By having shares of an MT, multiplication is performed as follows:

(1) Party i computes $\langle e\rangle_i^A = \langle x\rangle_i^A - \langle a\rangle_i^A$ and $\langle f\rangle_i^A = \langle y\rangle_i^A - \langle b\rangle_i^A$.
(2) Both parties communicate to reconstruct e and f.
(3) Party i computes its share of the multiplication as

$$\langle z\rangle_i^A = f \times \langle a\rangle_i^A + e \times \langle b\rangle_i^A + \langle c\rangle_i^A + i \times e \times f$$

For more complex operations, the function can be described as an Arithmetic circuit only consisting of addition and multiplication gates where in each step a single gate is processed accordingly.

3 RELATED WORK

Chameleon is essentially a two-party framework that uses a Semi-honest Third Party (STP) to generate correlated randomness in the offline phase. In the following, we review the use of third parties in secure computation as well as other secure two-party and multi-party computation frameworks.

Third Party-based Secure Computation. Regarding the involvement of a third party in secure two-party computation, there have been several works that consider an outsourcing or *server-aided* scenario, where the resources of one or more *untrusted* servers are employed to achieve sub-linear work in the circuit size of a function, even workload distribution, and output fairness. Realizing such a scenario can be done by either employing fully-homomorphic encryption (e.g., [5]) or extending Yao's garbled circuit protocol (e.g., [50]). Another important motivation for server-aided SFE is to address the issue of low-powered mobile devices, as done in [23–26, 34, 66]. Furthermore, server-aided secure computation can be used to achieve stronger security against active adversaries [43].

The secure computation framework of [46, Chapter 6] also utilizes correlated randomness. Beyond passive security and one STP, this framework also covers active security and multiple STPs.

GC-based Frameworks. The first implementation of the GC protocol is Fairplay [65] that allows users to write the program in a high-level language called Secure Function Definition Language (SFDL) which is translated into a Boolean circuit. FariplayMP [15] is the extension of Fairplay to the multiparty setting. FastGC [47] reduces the running time and memory requirements of the GC execution by using pipelining. TinyGarble [83] is one of the recent GC frameworks that proposes to generate compact and efficient Boolean circuits using industrial logic synthesis tools. TinyGarble also supports sequential circuits (cyclic graph representation of circuits) in addition to traditional combinational circuits (acyclic graph representation). ObliVM [61] provides a domain-specific programming language and a secure computation framework that facilitates the development process. Frigate [68] is a validated compiler and circuit interpreter for secure computation. Also, the authors of [68] test and validate several secure computation compilers and report the corresponding limitations. PCF (Portable Circuit Format) [53] has introduced a compact representation of Boolean circuits that enables better scaling of secure computation programs. Authors in [54] have shown the evaluation of a circuit with more than a billion gates in the malicious model by parallelizing operations.

Secret Sharing-based Frameworks. The Sharemind framework [18] is based on additive secret sharing over the ring $\mathbb{Z}_{2^{32}}$. The computation is performed with *three* nodes and is secure in the honest-but-curious adversary model where only one node can be corrupted. SEPIA [22] is a library for privacy-preserving aggregation of data for network security and monitoring. SEPIA is based on Shamir's secret sharing scheme where computation is performed by three (or more) privacy peers. VIFF (Virtual Ideal Functionality Framework) [31] is a framework that implements asynchronous secure computation protocols and is also based on Shamir's secret sharing. PICCO [87] is a source-to-source compiler that generates secure multiparty computation protocols from functions written in the `C` language. The output of the compiler is a `C` program that runs the secure computation using linear secret sharing. SPDZ [32] is a secure computation protocol based on additive secret sharing that is secure against $n-1$ corrupted computation nodes in the malicious model. Recent work of [3, 4, 39] introduces an efficient protocol for three-party secure computation. In general, for secret sharing-based frameworks, three (or more) computation nodes need to communicate in the online phase and in some cases, the communication is quadratic in the number of computation nodes. However, in Chameleon, the third node (STP) is not involved in the online phase which reduces the communication and running time.

While Chameleon offers more flexibility compared to secret-sharing based frameworks, it is also computationally more efficient compared to Sharemind and SEPIA: To perform each multiplication, Sharemind originally[4] needed 6 instances of the Du-Atallah protocol [18] while Chameleon needs 1 (when one operand is shared) or 2 (in the general case where both operands are shared). In SEPIA [22], all operations are performed modulo a prime number which is less efficient compared to modulo 2^l and also requires multiple multiplications for creating/reconstructing a share.

Mixed Protocol Frameworks. TASTY [42] is a compiler that can generate mixed protocols based on GC and HE. Several applications have been built that use mixed protocols, e.g., privacy-preserving ridge-regression [70], matrix factorization [70], iris and finger-code authentication [17], and medical diagnostics [9].

Recently, a new framework for compiling two-party protocols called EzPC [29] was presented. EzPC uses ABY as its cryptographic back-end: a simple and easy-to-use imperative programming language is compiled to ABY input. An interesting feature of EzPC is its "cost awareness", i.e. its ability to automatically insert type conversion operations in order to minimize the total cost of the resulting protocol. However, authors claim that ABY's GC engine always provides better performance for binary operations than GMW and thus convert only between A-SS and GC.

Our framework extends the ABY framework [35]. Specifically, we add support for signed fixed-point numbers which is essential for almost all machine learning applications such as processing deep neural networks. Our framework provides a faster online phase and a more efficient offline phase in terms of computation and communication due to the usage of an STP. Moreover, we implement a highly efficient vector dot product protocol based on correlated randomness generated by an STP.

Automatic Protocol Selection. The authors of [51] propose two methods, one heuristic and one based on integer programming, to find an optimal combination of two secure computation protocols, GC and HE. This methodology has been applied to the ABY framework in CheapSMC [73]. The current version of Chameleon does not provide automatic protocol selection. However, the methods of [29, 51, 73] can be applied in future work in order to automatically partition Chameleon programs.

Generation of Multiplication Triplets. Very recently, Lu and Sakuma [49] presented an efficient protocol for generating MTs that

[4]Sharemind replaced the Du-Atallah protocol with a new three-party multiplication protocol [19]. Due to its symmetry, we cannot modify this protocol to work with only two parties in the online phase as we do for the Du-Atallah protocol in §5.2.

are specially crafted for matrix multiplications by using additively shared matrices. The protocol results in a significant performance improvement in the offline phase compared to prior work, e.g., up to 110x faster run-time compared to SecureML [67] and MiniONN [62]. However, this protocol is limited to matrix multiplications, whereas Chameleon is generic and thus efficient for any operation.

4 THE CHAMELEON FRAMEWORK

Chameleon comprises of an *offline phase* and an *online phase*. The online phase is a two-party execution model that is run between two parties who wish to perform secure computation on their data. In the offline phase, a Semi-honest Third Party (STP) creates *correlated randomness* together with random seeds and provides it to the two parties as suggested in [46]. We describe how the STP can be implemented in §4.3 and its role in §5.2.

The online phase itself consists of three execution environments: GC, GMW, and Additive Secret Sharing (A-SS). We described the functionality of the GC and GMW protocols in §2 and we detail our implementations of these protocols in §5.1. We implement two different protocols for the multiplication operation on additive shares: a protocol based on Multiplication Triples (MTs) that we described in §2.4 and an optimized version of the Du-Atallah (DA) protocol [37] (cf. §5.2). In §4.1, we explain how the online phase works. In order to support highly efficient secure computations, all operations that do not depend on the run-time variables are shifted to the offline phase. The only cryptographic operations in the online phase are the Advanced Encryption Standard (AES) operations that are used in GC for which dedicated hardware acceleration is available in many processors via the AES-NI instruction set.

The offline phase includes four tasks: (i) precomputing all required OTs that are used in GC and type conversion protocols, thereby providing a very fast *encryption-free* online phase for OT, (ii) precomputing Arithmetic Multiplication Triples (A-MTs) used in the multiplication of additive secret shares, (iii) precomputing Boolean Multiplication Triples (B-MTs) used in the GMW protocol, and lastly, (iv) precomputing vector dot product shares (VDPS) used in the Du-Atallah protocol [37]. In order to reduce the communication in the offline phase from the STP to the two parties, we use the seed expansion technique [34] for generating A-MTs and B-MTs (cf. §5.4). We also introduce a novel technique that reduces the communication for generating VDPS (cf. §5.2).

4.1 Chameleon Online Execution Flow

In this section, we provide a high-level description of the execution flow of the online phase. As discussed earlier, linear operations such as ADD, SUB, and MULT are executed in A-SS. The dot product of two vectors of size n is also executed in A-SS which comprises n MULTs and $n-1$ ADDs. Non-linear operations such as CMP, EQ, MUX and bitwise XOR, AND, OR operations are executed in the GMW or GC protocol depending on which one is more efficient. Recall that in order to execute a function using the GMW or GC protocol, the function has to be described as a Boolean circuit.

However, the most efficient Boolean circuit description of a given function is different for the GMW and the GC protocol: In the GC protocol, the computation and communication costs only depend on the *total number of AND gates* (N_{AND}) in the circuit. Regardless of the number of XOR gates, functionality, and depth of the circuit, GC executes in a *constant* number of rounds. Communication is a linear function of the number of AND gates ($2 \times k \times N_{AND}$). Due to the Half-Gates optimization (cf. §5.1), computation is bounded by constructing the garbled tables (four fixed-key AES encryptions) and evaluating them (two fixed-key AES encryptions). The GMW protocol, on the other hand, has a different computation and communication model. It needs only bit-level AND and XOR operations for the computation, but one round of communication is needed per layer of AND gates. Therefore, the most efficient representation of a function in the GMW protocol is the one that has minimum *circuit depth*, more precisely, the minimum number of sequentially dependent layers of AND gates. As a result, when the network latency or the depth of the circuit is high, we use GC to execute non-linear functions, otherwise, GMW will be utilized. The computation and communication costs for atomic operations are given in App. C.

The program execution in Chameleon is described as different layers of operations where each layer is most efficiently realized in one of the execution environments. The execution starts from the first layer and the corresponding execution environment. Once all operations in the first layer are finished, Chameleon switches the underlying protocol and continues the process in the second execution environment. Changing the execution environment requires that the type of the shared secrets should be changed in order to enable the second protocol to continue the process. One necessary condition is that the cost of the share type translation must not be very high to avoid diminishing the efficiency achieved by the hybrid execution. For converting between the different sharing types, we use the methods from the ABY framework [35] which are based on highly efficient OT extensions.

Communication Rounds. The number of rounds that both parties need to communicate in Chameleon depends on the number of switches between execution environments and the depth of the circuits used in the GMW protocol. We want to emphasize that the number of communication rounds does not depend on the size of input data. Therefore, the network latency added to the execution time is quickly amortized over a high volume of input data.

4.2 Security Model

Chameleon is secure against honest-but-curious (HbC), a.k.a. semi-honest, adversaries. This is the standard security model in the literature and considers adversaries that follow the protocol but attempt to extract more information based on the data they receive and process. Honest-but-curious is the security model for the great majority of prior art, e.g., [18, 35, 83].

The Semi-honest Third Party (STP) can be either implemented using a physical entity, in a distributed manner using MPC among multiple non-colluding parties, using trusted hardware (hardware security modules or smartcards [34]), or using trusted execution environments such as Intel SGX [7]. In case the STP is implemented as a separate physical computation node, our framework is secure against semi-honest adversaries with an honest majority. The latter is identical to the security model considered in Sharemind [18]. In §3, we list further works based on similar assumptions. Please note that we introduce a new and more practical *computational* model that is superior to Sharemind since only two primary parties are involved in the online execution. This results in a significantly faster run-time while better matching real-world requirements.

4.3 Semi-honest Third Party (STP)

In Chameleon, the STP is only involved in the offline phase in order to generate correlated randomness [46]. It is not involved in the online phase and thus does not receive any information about the two parties' inputs nor the program being executed. The only exception is when computing VDPS for the Du-Atallah protocol: the STP needs to know the size of the vectors in each dot product beforehand. Since the security model in Chameleon is HbC with honest majority, some information can be revealed if the STP colludes with either party.

In order to prevent the STP from observing communication between the two parties, authenticated encryption is added to the communication channel. Also the communication between the STP and the two parties is encrypted, so they cannot reconstruct the other party's private inputs from observed messages.

5 CHAMELEON DESIGN AND IMPLEMENTATION

In this section, we provide a detailed description of the different components of Chameleon. Chameleon is written in C++ and accepts the program written in C++. The implementation of the GC and GMW engines is covered in §5.1 and the A-SS engine is described in §5.2. §5.3 illustrates how Chameleon supports signed fixed-point representation. The majority of cryptographic operations is shifted from the online to the offline phase. Thus, in §5.4, we describe the process of generating Arithmetic/Boolean Multiplication Triples (A-MTs/B-MTs). §5.5 provides our STP-based implementation for fast Oblivious Transfer and finally the security justification of Chameleon is given in §5.6.

5.1 GC and GMW Engines

Chameleon's implementation of the GC and GMW protocol is based on ABY [35]. Therefore, the input to the engines is the topologically sorted list of Boolean gates in the circuit as an .aby file. The GC engine includes the most recent optimizations: Free-XOR [52], fixed-key AES garbling [14], and Half-Gates [86]. We synthesized GC-optimized circuits for many primitive functions. Likewise, for the GMW engine all circuits are depth-optimized as described in [33] to incur the least latency during the protocol execution. A user can simply use these circuits by calling regular functions in C++.

5.2 A-SS Engine

In Chameleon, linear operations, i.e., ADD, SUB, MULT, are performed using additive secret sharing in the ring $\mathbb{Z}_{2^l}$. We discussed in §2.4 how to perform a single MULT using a multiplication triple. However, there are other methods to perform a MULT: (i) The protocol of [16] has very low communication in the online phase. However, in contrast to our computation model, it requires STP interaction with the other two parties in the online phase. (ii) The Du-Atallah protocol [37] is another method to perform multiplication on additive shared values which we describe next.

The Du-Atallah Multiplication Protocol [37]. In this protocol, two parties $\mathcal{P}_0$ (holding x) and $\mathcal{P}_1$ (holding y) together with a third party $\mathcal{P}_2$ can perform the multiplication $z = x \times y$. At the end of this protocol, z is additively shared between all *three* parties. The protocol works as follows:

(1) $\mathcal{P}_2$ randomly generates $a_0, a_1 \in_R \mathbb{Z}_{2^l}$ and sends a_0 to $\mathcal{P}_0$ and a_1 to $\mathcal{P}_1$.
(2) $\mathcal{P}_0$ computes $(x + a_0)$ and sends it to $\mathcal{P}_1$. Similarly, $\mathcal{P}_1$ computes $(y + a_1)$ and sends it to $\mathcal{P}_0$.
(3) $\mathcal{P}_0, \mathcal{P}_1$, and $\mathcal{P}_2$ can compute their share as $\langle z\rangle_0^A = -a_0 \times (y + a_1)$, $\langle z\rangle_1^A = y \times (x + a_0)$, and $\langle z\rangle_2^A = a_0 \times a_1$, respectively.

It can be observed that the results are true additive shares of z: $\langle z\rangle_0^A + \langle z\rangle_1^A + \langle z\rangle_2^A = z$. Please note that this protocol computes shares of a multiplication of two numbers held by two parties in *cleartext*. In the general case, where both x and y are additively shared between two parties ($\mathcal{P}_0$ holds $\langle x\rangle_0^A$, $\langle y\rangle_0^A$ and $\mathcal{P}_1$ holds $\langle x\rangle_1^A$, $\langle y\rangle_1^A$), the multiplication can be computed as $z = x \times y = (\langle x\rangle_0^A + \langle x\rangle_1^A) \times (\langle y\rangle_0^A + \langle y\rangle_1^A)$. The two terms $\langle x\rangle_0^A \times \langle y\rangle_0^A$ and $\langle x\rangle_1^A \times \langle y\rangle_1^A$ can be computed locally by $\mathcal{P}_0$ and $\mathcal{P}_1$, respectively. *Two* instances of the Du-Atallah protocol are needed to compute shares of $\langle x\rangle_0^A \times \langle y\rangle_1^A$ and $\langle x\rangle_1^A \times \langle y\rangle_0^A$. Please note that $\mathcal{P}_i$ should not learn $\langle x\rangle_{1-i}^A$ and $\langle y\rangle_{1-i}^A$, otherwise, secret values x and/or y are revealed to $\mathcal{P}_i$. At the end, $\mathcal{P}_0$ has

$$\langle x\rangle_0^A \times \langle y\rangle_0^A,\ \left\langle \langle x\rangle_0^A \times \langle y\rangle_1^A \right\rangle_0^A,\ \left\langle \langle x\rangle_1^A \times \langle y\rangle_0^A \right\rangle_0^A$$

and $\mathcal{P}_1$ has

$$\langle x\rangle_1^A \times \langle y\rangle_1^A,\ \left\langle \langle x\rangle_0^A \times \langle y\rangle_1^A \right\rangle_1^A,\ \left\langle \langle x\rangle_1^A \times \langle y\rangle_0^A \right\rangle_1^A,$$

where $\langle z\rangle_0^A$, $\langle z\rangle_1^A$ are the summations of each party's shares.

The Du-Atallah protocol was used in Sharemind [18] where there are three active computing nodes that are involved in the online phase, whereas, in Chameleon, the third party (STP) is only involved in the offline phase. This problem can be solved since the role of $\mathcal{P}_2$ can be shifted to the offline phase as follows: (i) Step one of the Du-Atallah protocol can be computed in the offline phase for as many multiplications as needed. (ii) In addition, $\mathcal{P}_2$ randomly generates another l-bit number a_2 and computes $a_3 = (a_0 \times a_1) - a_2$. $\mathcal{P}_2$ sends a_2 to $\mathcal{P}_0$ and a_3 to $\mathcal{P}_1$ in the offline phase. During the online phase, both parties additionally add their new shares (a_2 and a_3) to their shared results: $\langle z\rangle_{0,new}^A = \langle z\rangle_0^A + a_2$ and $\langle z\rangle_{1,new}^A = \langle z\rangle_1^A + a_3$.

Security. This modification is perfectly secure since $\mathcal{P}_0$ has received a true random number and $\mathcal{P}_1$ has received a_3 which is an additive share of $(a_0 \times a_1)$. Since a_2 has uniform distribution, the probability distribution of a_3 is also uniform [18] and as a result, $\mathcal{P}_1$ cannot infer additional information.

Du-Atallah Protocol with one cleartext operand. As we will discuss in §6, in many cases, the computation model is such that one operand x is held in cleartext by one party, e.g., $\mathcal{P}_0$, and the other operand y is shared among two parties: $\mathcal{P}_0$ has $\langle y\rangle_0^A$ and $\mathcal{P}_1$ has $\langle y\rangle_1^A$. This situation repeatedly arises when the intermediate result is multiplied by one of the party's inputs which is not shared. In this case, only one instance of the Du-Atallah protocol is needed to compute $x \times \langle y\rangle_1^A$. As analyzed in this section, employing this variant of the Du-Atallah protocol is more efficient than the protocol based on MTs. Please note that in order to utilize MTs, both operands need to be shared among the two parties first, which, as we argue here, is inefficient and unnecessary. Tab. 1 summarizes

Table 1: Summary of properties of the Du-Atallah multiplication protocol and the protocol based on Multiplication Triples in §2.4. (i, j) means $\mathcal{P}_0$ and $\mathcal{P}_1$ have to perform i and j multiplications in cleartext, respectively. Offline and online communication costs are expressed in number of bits. Online communication costs correspond to data transmission in each direction. *Initial sharing of x is also considered.

Protocol	# MULT ops	Online Comm.	Offline Comm.	Rounds
Multiplication Triple	(3,4)	$2 \cdot l$	$3 \cdot l$	2^*
Du-Atallah	(1,2)	l	$2 \cdot l$	1

the computation and communication costs for the Du-Atallah protocol and the protocol based on MTs (§2.4). As can be seen, online computation and communication is improved by factor 2x. Also, the offline communication is improved by factor 3x. Unfortunately, using the Du-Atallah protocol in this format will reduce the efficiency of vector dot product computation in Chameleon. Please note that it is no longer possible to perform a complete dot product of two vectors by two parties only. The reason is that the third share ($\langle z \rangle_2^A = a_0 \times a_1$) is shared between two parties ($\mathcal{P}_0$ and $\mathcal{P}_1$). However, this problem can be solved by a modification which we describe next.

Du-Atallah Protocol and Vector Dot Product. We further modify the optimized Du-Atallah protocol such that the complete vector dot product is efficiently processed. The idea is that instead of the STP additively sharing its shares, it first sums its shares and then sends the additively shared versions to the two parties. Consider vectors of size n. The STP needs to generate n different a_0 and a_1 as a list for a single vector multiplication. We denote the j^{th} member of the list as $[a_0]_j$ and $[a_1]_j$. Our modification requires that the STP generates a single l-bit value a_2 and sends it to $\mathcal{P}_0$. The STP also computes $a_3 = \sum_{i=0}^{n-1} [a_0]_j \times [a_1]_j - a_2$ and sends it to $\mathcal{P}_1$. We call a_2 and a_3 the *Vector Dot Product Shares* (VDPS). This requires that the STP knows the size of the array in the offline phase. Since the functionality of the computation is not secret, we can calculate the size and number of all dot products in the offline phase and ask for the corresponding random shares from the STP.

Reducing Communication. A straightforward implementation of the offline phase of the Du-Atallah protocol requires that the STP sends $\sim n$ random numbers of size l ($[a_0]_j$ and $[a_1]_j$) to $\mathcal{P}_0$ and $\mathcal{P}_1$ for a single dot product of vectors of size n. However, we suggest reducing the communication using a Pseudo Random Generator (PRG) for generating the random numbers as was proposed in [34]. Instead of sending the complete list of numbers to each party, the STP can create and send random PRG seeds for each string to the parties such that each party can create $[a_0]_j$ and $[a_1]_j$ locally using the PRG. For this purpose, we implement the PRG using Advanced Encryption Standard (AES), a low-cost block cipher, in counter mode (AES CTR-DRBG). Our implementation follows the description of the NIST Recommendation for DRBGs [8]. From a 256-bit seed, AES CTR-DRBG can generate 2^{63} indistinguishable random bits. If more than 2^{63} bits are needed, the STP sends more seeds to the parties. The STP uses the same seeds in order to generate a_2 and a_3 for each dot product. Therefore, the communication is reduced from $n \times l$ bits to sending a one-time 256-bit seed and an l-bit number per single dot product.

Performance Evaluation. We give an empirical performance evaluation of our optimized VDP protocol in App. B: the evaluated SVM classification mainly consists of a VDP computation together with a negligible subtraction and comparison operation.

5.3 Supporting Signed Fixed-point Numbers

Chameleon supports Signed Fixed-point Numbers (SFN) in addition to integer operations. Supporting SFN requires not only that all three secure computation protocols (GC, GMW, and Additive SS) support SFN but also the secret translation protocols to be compatible. We note that the current version of the ABY framework only supports unsigned integers and IEEE 754 floating point numbers [33]. We added an abstraction layer to the ABY framework such that it supports SFN.

All additive secret sharing protocols only support unsigned integer values. However, in this section, we describe how such protocols can be modified to support *signed fixed-point* numbers. Supporting *signed integers* can be done by representing numbers in two's complement format. Consider the ring $\mathbb{Z}_{2^l}$ which consists of unsigned integer numbers $\{0, 1, 2, \dots, 2^{l-1} - 1, 2^{l-1}, \dots, 2^l - 1\}$. We can perform signed operations by simply *interpreting* these numbers as the two's complement format: $\{0, 1, 2, \dots, 2^{l-1} - 1, -2^{l-1}, \dots, -1\}$. By doing so, signed operations work seamlessly.

In order to support fixed-point precision, one solution is to interpret signed integers as signed fixed-point numbers. Each number is represented in two's complement format with the Most Significant Bit (MSB) being the sign bit. There are α and β bits for integer and fraction parts, respectively. Therefore, the total number of bits is equal to $\gamma = 1 + \alpha + \beta$. While this works perfectly for addition and subtraction, it cannot be used for multiplication. The reason is that when multiplying two numbers in a ring, the rightmost $2 \times \beta$ bits of the result correspond to the fraction part while β bits of the MSBs are overflown and discarded. Our solution to this problem is to perform all operations in the ring $\mathbb{Z}_{2^l}$ where $l = \gamma + \beta$. After each multiplication, we shift the result β bits to the right while replicating the sign bit for β MSBs. While bitshifting by a constant and sign bit replication is essentially free in GC/GMW, it is non-trivial in additive sharing. Thus, a conversion from additive sharing to GC/GMW and back is required between multiplications. Compared to [67], where the authors apply a similar approach to fixed-point arithmetic but simply truncate additive shares, this prevents introducing up to 1 bit inaccuracy per multiplication. The overhead for the conversions is given in Tab. 9 in App. C. These additional costs are certainly smaller than adapting the approach of [77], where the authors naively apply the same method as used for floating-point arithmetic in ABY [33], i.e., they use hardware compilers to generate circuits which perform fixed-point arithmetic in GC. Following the observation that in GC the overhead for multiplication even for integer numbers is large (cf. Tabs. 7 and 8), we expect our mixed-protocol approach to greatly outperform their implementation. Please note that for the machine learning applications discussed in §6 actually no preventable overhead for protocol conversion occurs: between all multiplications a non-linear function is computed, which requires conversion to GC/GMW anyhow.

This assumes that in addition to the support by the computation engines, share translation protocols actually work correctly.

Share translation from GC to GMW works fine as it operates on bit-level and is transparent to the number representation format. Share translation from GC/GMW to additive sharing either happens using a subtraction circuit or OT. In the first case, the result is valid since the subtraction of two signed fixed-point numbers in two's complement format is identical to subtracting two unsigned integers. In the second case, OT is on bit-level and again transparent to the representation format. In Chameleon, as in ABY, we use the OT method for share translation from GC/GMW to additive due to reduced complexity. Finally, share translation from additive sharing to GC/GMW is correct because it uses an addition circuit, which is identical for unsigned integers and signed fixed-point numbers.

Floating Point Operations. Chameleon supports floating point operations by performing all computations in the GC or GMW protocol as described in [33] for ABY. A future direction of this work can be to break down the primitive floating point operations, e.g., ADD, MULT, SUB, etc. into smaller atomic operations based on integer values. Consequently, one can perform the linear operations in the ring and non-linear operations in GC/GMW, providing a faster execution for floating-point operations.

Most methods for secure computation on floating and fixed point numbers proposed in the literature were realized in Shamir's secret sharing scheme, e.g. [2, 27, 55, 74, 87], but some of them also in GC [74], GMW [33], and HE [63] based schemes. The quality of the algorithms varies from self-made to properly implemented IEEE 754 algorithms, such as in [33, 74]. The corresponding software implementations were done either in the frameworks Sharemind [18] and PICCO [87], or as standalone applications. For fixed-point arithmetics, Aliasgari et al. [2] proposed algorithms that outperform even integer arithmetic for certain operations. As a future direction of this work, we plan to integrate their methodology in Chameleon.

5.4 Generating Multiplication Triples

As we discussed in §2.4, each multiplication on additive secret shares requires an Arithmetic Multiplication Triple (A-MT) and one round of communication. Similarly, evaluating each AND gate in the GMW protocol requires a Boolean Multiplication Triple (B-MT) [34]. In the offline phase, we calculate the number of MTs ($N_{\text{A-MT}}$ and $N_{\text{B-MT}}$). The STP precomputes all MTs needed and sends them to both parties. More precisely, to generate A-MTs, the STP uses a PRG to produce five l-bit random numbers corresponding to a_0, b_0, c_0, a_1, and b_1. We denote the j^{th} triple with $[.]_j$. Therefore, the STP completes MTs by computing c_1's as $[c_1]_j = ([a_0]_j + [a_1]_j) \times ([b_0]_j + [b_1]_j) - [c_0]_j$. Finally, the STP sends $[a_0]_j, [b_0]_j$, and $[c_0]_j$ to the first party and $[a_1]_j, [b_1]_j$, and $[c_1]_j$ to the second party for $j = 1, 2, ..., N_{\text{A-MT}}$. Computing B-MTs is also very similar with the only differences that all numbers are 1-bit and $[c_1]_j$ is calculated as $[c_1]_j = ([a_0]_j \oplus [a_1]_j) \land ([b_0]_j \oplus [b_1]_j) \oplus [c_0]_j$.

Reducing Communication. A basic implementation of precomputing A-MTs and B-MTs requires communication of $3 \times l \times N_{\text{A-MT}}$ and $3 \times N_{\text{B-MT}}$ bits from the STP to each party, respectively. However, similar to the idea of [34] presented in §5.2, we use a PRG to generate random strings from seeds locally for each party. To summarize the steps: the STP

(1) generates two random seeds: seed_0 for generating $[a_0]_j, [b_0]_j$, and $[c_0]_j$ and seed_1 for $[a_1]_j$ and $[b_1]_j$;
(2) computes $[c_1]_j = ([a_0]_j + [a_1]_j) \times ([b_0]_j + [b_1]_j) - [c_0]_j$ for $j = 1, 2, ..., N_{A-MT}$;
(3) sends seed_0 to the first party and seed_1 together with the list of $[c_1]_j$ to the second party.

After receiving the seeds, both parties locally generate their share of the triples using the same PRG. This method reduces the communication from $3 \times l \times N_{\text{A-MT}}$ to 256 and $256 + l \times N_{\text{A-MT}}$ bits for the first and second party, respectively. The STP follows a similar process to generate B-MTs. Fig. 1 illustrates the seed expansion idea to generate MTs [34].

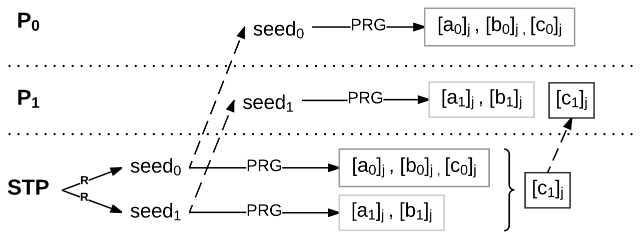

Figure 1: Seed expansion process to precompute A-MTs/B-MTs with low communication.

5.5 Fast STP-aided Oblivious Transfer

Utilizing the idea of correlated randomness [46], we present an efficient and fast protocol for Oblivious Transfer that is aided by the Semi-honest Third Party (STP). Our protocol comprises an offline phase (performed by the STP) and an online phase (performed by the two parties). The protocol is described for one 1-out-of-2 OT. The process repeats for as many OTs as required. In the offline phase, the STP generates random masks q_0, q_1 and a random bit r and sends q_0, q_1 to the sender and r, q_r to the receiver. In the online phase, two parties execute the online phase of Beaver's OT precomputation protocol [12]. Please note that all OTs in Chameleon including OTs used in GC and secret translation from GC/GMW to Additive are implemented as described above.

Reducing Communication. Similar to the idea discussed in §5.4, the STP does not actually need to send the list of (q_0, q_1) to the sender and r to the receiver. Instead, it generates two random seeds and sends them to the two parties. The STP only needs to send the full list of q_r to the receiver.

5.6 Security

Chameleon is based on the ABY framework [35] where we replace the interactive offline phase with the following STP-based protocols: (i) STP-aided OTs (cf. §5.5) are implemented via Beaver's OT precomputation [12] where the original OTs are sent by the STP, which is trivially secure. (ii) STP-aided generation of MTs (cf. §5.4) was proven secure in [34, 46]. (iii) STP-aided multiplication (cf. §5.2) is done based on an STP-aided extension of the Du-Atallah multiplication protocol [37] for which we have argued security already in §5.2; all further optimizations are simply a compression of the data sent by the STP and hence do not leak any additional information. In summary, security of Chameleon follows from the security of ABY and the security of our STP-based protocols, so we can state the following theorem.

Theorem 5.1. *Chameleon's STP-based protocols are secure against HbC adversaries under the assumption that at most one of the two parties is passively corrupted and none of them colludes with the STP.*

6 MACHINE LEARNING APPLICATIONS

Many applications can benefit from our framework since it is generic. However, due to Chameleon's optimized VDP protocol and signed fixed-point number support, especially the efficiency of machine learning tasks can be improved. In particular, we show how Chameleon can be leveraged in Deep Learning (cf. §6.1) and classification based on SVMs (cf. App. B), and compare its performance to previous works. In App. D we review further works on privacy-preserving machine learning.

We run our experiments for long-term security parameters (128-bit security) on machines equipped with Intel Core i7-4790 CPUs @ 3.6 GHz and 16 GB of RAM with AES-NI support. The STP is instantiated as a separate compute node running a `C/C++` implementation. The communication between the STP and its clients as well as between the clients is protected by TLS with client authentication. Except when stated otherwise, all parties run on different machines within the same Gigabit network.

6.1 Deep Learning

We evaluate our framework on Deep Neural Networks (DNNs) and a more sophisticated variant, Convolutional Deep Neural Networks (CNNs). Processing both, DNNs and CNNs, requires the support of signed fixed-point numbers. We compare our results with the state-of-the-art Microsoft CryptoNets [36], which is a customized solution for this purpose based on homomorphic encryption, as well as other recent solutions.

Deep Neural Networks. Deep learning is a very powerful method for modeling and classifying raw data that has gained a lot of attention in the past decade due to its superb accuracy. Deep Learning automatically learns complex features using artificial neural networks. While there are many different DNNs and CNNs, they all share a similar structure: They are networks of multiple layers stacked on top of each other where the output of each layer is the input to the next layer. The input to DNNs is a feature vector, which we denote as $\mathbf{x}$. The input is passed through the intermediate layers (hidden layers). The output vector of the L^{th} layer is $\mathbf{x}^{(L)}$ where $x_i^{(L)}$ denotes the i^{th} element. The length of the vector can change after each layer. The length of the intermediate result vector at layer L is $N_L = \text{length}(\mathbf{x}^{(L)})$.

A DNN is composed of a series of different layers. (i) *Fully Connected layer (FC)*: the output $\mathbf{x}^{(L)}$ is the matrix multiplication of input vector $\mathbf{x}^{(L-1)}$ and a matrix weight $\mathbf{W}$, that is, $\mathbf{x}^{(L)} = \mathbf{x}^{(L-1)} \cdot \mathbf{W}^{(L)}$. In general, the size of the input and output of the *FC* layer is denoted as $FC^{N_{L-1} \times N_L}$. (ii) *Activation layer (Act)*: applies an activation function $f(.)$ on the input vector, i.e., $x_i^{(L)} = f(x_i^{(L-1)})$. The activation function is usually a Rectified Linear Unit (ReLu), Tangent-hyperbolic (Tanh), or Sigmoid function [36, 81].

The input to a CNN is a picture represented as a matrix $\mathbf{X}$ where each element corresponds to the value of a pixel. Pictures can have multiple color channels, e.g., RGB, in which case the picture is represented as a multidimensional matrix, a.k.a., a *tensor*. CNNs are similar to DNNs but they can potentially have additional layers: (i) *Convolution layer (C)*: essentially a weighted sum of a "square region" of size s_q in the proceeding layer. To compute the next output, the multiplication window on the input matrix is moved by a specific number, called stride (s_t). The weight matrix is called *kernel*. There can be N_{map} (called map count) kernels in the convolution layer. (ii) *Mean-pooling (MeP)*: the average of each square region of the proceeding layer. (iii) *Max-pooling (MaP)*: the maximum of each square region of the proceeding layer. The details of all layers are provided in Tab. 2.

Many giant technology companies such as Google, Microsoft, Facebook, and Apple have invested millions of dollars in accurately training neural networks to serve in different services. Clients who want to use these services currently need to reveal their inputs, which may contain sensitive information, to cloud servers. Thus, there is a special need to run a neural network (trained by the cloud server) on input from another party (clients) while keeping both the network parameters and the input private. For this purpose, Microsoft has announced CryptoNets [36] which can process encrypted queries in neural networks using homomorphic encryption. Next, we compare the performance result of Chameleon to CryptoNets and other more recent works.

Table 2: Different types of layers in DNNs and CNNs.

Layer	Functionality
FC	$x_i^{(L)} = \sum_{j=0}^{N_{L-1}-1} W_{ij}^{(L-1)} \times x_j^{(L-1)}$
Act	$x_i^{(L)} = f(x_i^{L-1})$
C	$x_{ij}^{(L)} = \sum_{a=0}^{s_q-1} \sum_{b=0}^{s_q-1} W_{ab}^{(L-1)} \times x_{(i \cdot s_t+a)(j \cdot s_t+b)}^{L-1}$
MeP	$x_{ij}^{(L)} = \text{Mean}(x_{(i+a)(j+b)}^{L-1}),\ a,b \in \{1,2,...,s_q\}$
MaP	$x_{ij}^{(L)} = \text{Max}(x_{(i+a)(j+b)}^{L-1}),\ a,b \in \{1,2,...,s_q\}$

Comparison with Previous Works (MNIST Dataset). We compare the performance of Chameleon when classifying images from the MNIST dataset to recent works performing the same task in Tab. 3. The MNIST dataset [59] contains 60,000 images of handwritten digits. Each image is represented as 28×28 pixels with values between 0 and 255 in gray scale.

We train a CNN architecture using the Keras library [30] running on top of TensorFlow [1] using 50,000 images. We achieve a test accuracy of ~99 % examined over 10,000 test images. The architecture of the trained CNN is depicted in Fig. 2 and composed of: (i) *C* layer with a kernel of size 5×5, stride 2, and map count 5. (ii) *Act* layer with ReLu as the activation function. (iii) $FC^{980 \times 100}$ layer. (iv) Another ReLu *Act* layer, and (v) a $FC^{100 \times 10}$ layer. The output of the last layer is a vector of ten numbers where each number represents the probability of the image being each digit (0-9). We extract the maximum value and output it as the classification result.

The implementation of the CNN architecture in Chameleon is straightforward: C and FC layers are implemented according to their specification in Tab. 2 making use of our efficient VDP protocol (cf. §5.2). The ReLU *Act* layer is efficiently implemented as a MUX operation on the sign bit in GMW. The final arg-max operation is already built-in in ABY and evaluates a balanced binary

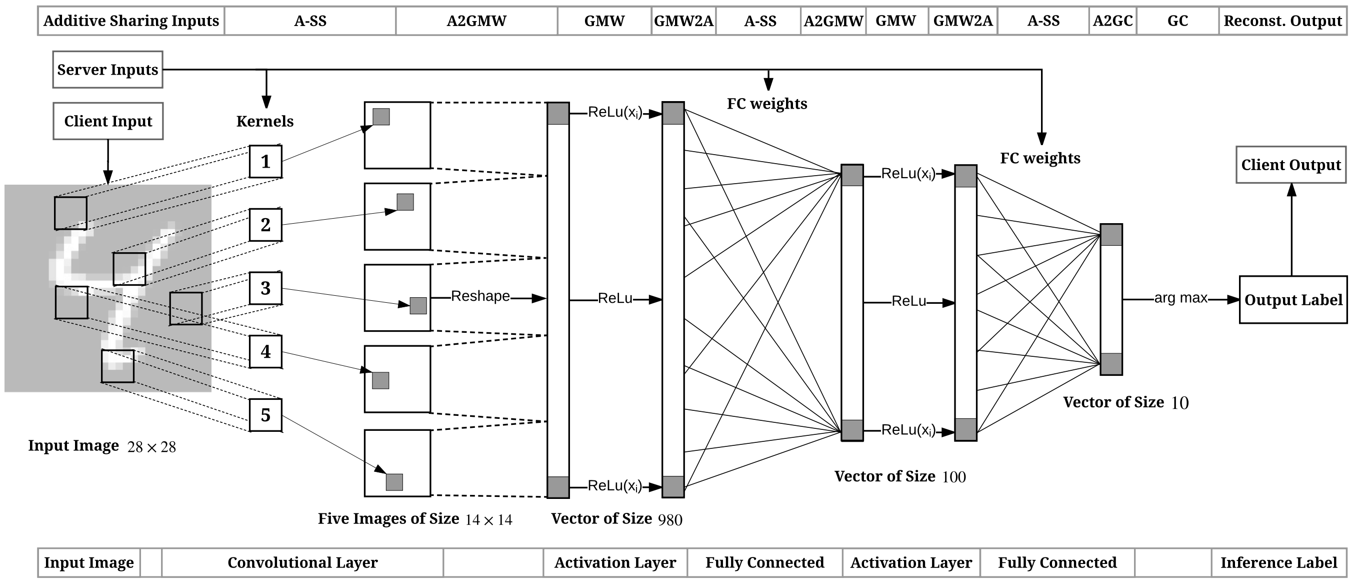

Figure 2: Architecture of our Convolutional Neural Network trained for the MNIST dataset. The upper bar illustrates which protocol is being executed at each phase of the CNN. The lower bar shows different layers of the CNN from the DL perspective.

Table 3: Comparison of secure deep learning frameworks, their characteristics, and performance results for classifying one image from the MNIST dataset in the LAN setting.

Framework	Methodology	Non-linear Activation and Pooling Functions	Classification Timing (s)			Communication (MB)			Classification Accuracy
			Offline	Online	Total	Offline	Online	Total	
Microsoft CryptoNets [36]	Leveled HE	✗	-	-	297.5	-	-	372.2	98.95 %
DeepSecure [77]	GC	✓	-	-	9.67	-	-	791	99 %
SecureML [67]	Linearly HE, GC, SS	✗	4.70	0.18	4.88	-	-	-	93.1 %
MiniONN (Sqr Act.) [62]	Additively HE, GC, SS	✗	0.90	0.14	1.04	3.8	12	15.8	97.6 %
MiniONN (ReLu + Pooling) [62]	Additively HE, GC, SS	✓	3.58	5.74	9.32	20.9	636.6	657.5	99 %
EzPC [29]	GC, Additive SS	✓	-	-	5.1	-	-	501	99 %
Chameleon (This Work)	GC, GMW, Additive SS	✓	1.25	0.99	2.24	5.4	5.1	10.5	99 %

tree in GC consisting of comparison and MUX gates. The server's input consists of the kernels' values and FC weights whereas the client's input is the image to be classified. The output of the secure computation is the classification (inference) label. The lower bar in Fig. 2 shows the order of the different layers of the CNN. The upper bar depicts the corresponding protocol that executes the current part of the CNN and also the required conversions. Additionally, the figure shows the sizes of matrices and vectors in each step.

The performance results compared with Microsoft CryptoNets and other recent works are provided in Tab. 3. The table further shows differences in the employed methodologies and the support of non-linear activation and pooling functions. Except for DeepSecure [77], the structure of the CNN architectures evaluated in the other works differs. However, the relevant measures here are classification accuracy and performance. Since our accuracy of 99 % equals the highest achieved by competitors, the following performance comparison is fair for frameworks with equal accuracy and to our disadvantage for frameworks with less accuracy. More specific differences are discussed below.

We report our run-time as Offline/Online/Total. As can be seen, Chameleon is 133x faster compared to the customized solution based on homomorphic encryption of CryptoNets [36]. They performed the experiments on a similar machine (Intel Xeon ES-1620 CPU @ 3.5 GHz with 16 GB of RAM). Please note that in CryptoNets [36] numbers are represented with 5 to 10 bit precision while in Chameleon all numbers are represented as 64 bit numbers. Although the precision does not considerably change the accuracy for the MNIST dataset, it might significantly reduce the accuracy results for other datasets. In addition, the CryptoNets framework neither supports non-linear activation nor pooling functions. However, it is worth-mentioning that CryptoNets can process a batch of images of size 8,192 with no additional costs. Therefore, the CryptoNets framework can process up to 51,739 predictions per hour. Nonetheless, it is necessary that the system batches a large number of images and processes them together. This, in turn, might reduce the throughput of the network significantly.

A recent solution based on leveled homomorphic encryption is called CryptoDL [44]. In CryptoDL, several activation functions are approximated using low-degree polynomials and mean-pooling is used as a replacement for max-pooling. The authors state up to 163,840 predictions per hour for the same batch size as in CryptoNets. However, for a single instance, CryptoDL incurs the same

Table 4: Classification time (in seconds) and communication costs (in megabytes) of Chameleon for different batch sizes of the MNIST dataset in the WAN setting (100 Mbit/s bandwidth, 100 ms round-trip time).

	Classification Time (s)			Communication (MB)		
Batch Size	Offline	Online	Total	Offline	Online	Total
1	4.03	2.85	6.88	7.8	5.1	12.9
10	10.00	10.65	20.65	78.4	50.5	128.9
100	69.38	84.09	153.47	784.1	505.3	1289.4

computation and communication costs as for one batch. Also, note that in Chameleon one can implement and evaluate virtually any activation and pooling function.

The DeepSecure framework [77] is a GC-based framework for secure Deep Learning inference. DeepSecure also proposes data-level and network-level preprocessing steps before the secure computation protocol. They report a run-time of 9.67 s to classify images from the MNIST dataset using a CNN similar to CryptoNets. They utilize non-linear activation and pooling functions. Chameleon is 4.3x faster and requires 75x less communication compared to DeepSecure when running an identical CNN.

SecureML [67] is a framework for privacy-preserving machine learning. Similar to CryptoNets, SecureML focuses on linear activation functions. The MiniONN [62] framework reduces the classification latency on an identical network from 4.88 s to 1.04 s using similar linear activation functions. MiniONN also supports non-linear activation functions and max-pooling. They report a classification latency of 9.32 s while successfully classifying MNIST images with 99 % accuracy. For a similar accuracy and network, Chameleon has 4.2x lower latency and requires 63x less communication.

For the evaluation of the very recent EzPC framework [29], the authors implement the CNN from MiniONN in a high-level language. The EzPC compiler translates this into an ABY program while automatically inserting conversions between GC and A-SS. This results in a total run-time of 5.1 s for classifying one image. Chameleon requires 48x less communication.

Tab. 3 shows that the total run-time of the end-to-end execution of Chameleon for a single image is only 2.24 s. However, Chameleon can easily be scaled up to classify multiple images at the same time using a CNN with non-linear activation and pooling functions. For a batch size of 100, our framework requires only 0.18 s processing time and 10.5 MB communication per image providing up to 20,000 predictions per hour in the LAN setting. Tab. 4 furthermore shows the required run-times and communication for different batch sizes in a WAN setting where we restrict the bandwidth to 100 Mbit/s with a round-trip time of 100 ms. In the WAN setting, we replace all GMW protocol invocations with the GC protocol to benefit from its constant round property.

Comparison with Previous Works (CIFAR-10 Dataset). In accordance with previous works, we also evaluate our framework by running a CNN to classify images from the CIFAR-10 dataset [56]. The CIFAR-10 dataset comprises 60,000 color images with a resolution of 32 x 32 pixels. We implement and train a CNN with the same architecture as given in Fig. 13 in [62], which achieves 81.61 % accuracy. Compared to the CNN used for classifying MNIST images, the architecture of this CNN is more sophisticated: in total there

Table 5: Classification time (in seconds) and communication costs (in gigabytes) of secure deep learning frameworks for one image from the CIFAR-10 dataset in the LAN setting.

Framework	Classification Time (s)			Communication (GB)		
	Offline	Online	Total	Offline	Online	Total
MiniONN [62]	472	72	544	6.23	3.05	9.28
EzPC [29]	-	-	265.6	-	-	40.63
Chameleon (This Work)	22.97	29.7	52.67	1.21	1.44	2.65

are 7 convolution layers, 7 ReLu activation layers, 2 mean-pooling layers, and one fully connected layer. The exact CNN architecture is given in Fig. 3 in App. A. We report the performance results when classifying one image in Tab. 5. Compared to MiniONN [62], the total run-time is reduced by factor 10.3x. The more recent EzPC framework [29] is still by factor 5x slower than our solution and requires 15x more communication.

ACKNOWLEDGMENTS

We thank the anoynmous ASIACCS'18 reviewers and our shepherd Peeter Laud for helpful comments on our paper. The authors from UCSD are supported in parts by the National Science Foundation grant (CNS-1619261), Semiconductor Research Corporation grant (2016-TS-269), Office of Naval Research grant (N00014-17-1-2500), and NSF Trust-Hub grant (CNS-1649423). This work has also been co-funded by the DFG as part of project E4 within the CRC 1119 CROSSING and by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*.

[2] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. 2013. Secure Computation on Floating Point Numbers. In *NDSS*.

[3] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman, and O. Weinstein. 2017. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *IEEE S&P*.

[4] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *CCS*.

[5] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. 2012. Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. In *EUROCRYPT*.

[6] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *CCS*.

[7] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi. 2017. Secure multiparty computation from SGX. In *FC*.

[8] E. Barker and J. Kelsey. 2015. *NIST Special Publication 800-90A Revision 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* Technical Report.

[9] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. 2009. Secure evaluation of private linear branching programs with medical applications. In *ESORICS*.

[10] M. Barni, P. Failla, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. 2011. Privacy-Preserving ECG Classification With Branching Programs and Neural Networks. In *IEEE TIFS*.

[11] D. Beaver. 1991. Efficient multiparty protocols using circuit randomization. In *CRYPTO*.

[12] D. Beaver. 1995. Precomputing oblivious transfer. In *CRYPTO*.

[13] D. Beaver. 1996. Correlated pseudorandomness and the complexity of private computations. In *STOC*.

[14] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. 2013. Efficient garbling from a fixed-key blockcipher. In *IEEE S&P*.

[15] A. Ben-David, N. Nisan, and B. Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *CCS*.

[16] A. Ben-Efraim, Y. Lindell, and E. Omri. 2016. Optimizing Semi-Honest Secure Multiparty Computation for the Internet. In *CCS*.
[17] M. Blanton and P. Gasti. 2011. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS*.
[18] D. Bogdanov, S. Laur, and J. Willemson. 2008. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*.
[19] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. 2012. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security* 11, 6 (2012).
[20] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. 2015. Machine Learning Classification over Encrypted Data. In *NDSS*.
[21] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. 2007. Privacy-preserving remote diagnostics. In *CCS*.
[22] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. 2010. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *USENIX Security* (2010).
[23] H. Carter, C. Lever, and P. Traynor. 2014. Whitewash: outsourcing garbled circuit generation for mobile devices. In *ACSAC*.
[24] H. Carter, B. Mood, P. Traynor, and K. R. B. Butler. 2013. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *USENIX Security*.
[25] H. Carter, B. Mood, P. Traynor, and K. R. B. Butler. 2015. Outsourcing Secure Two-Party Computation as a Black Box. In *CANS*.
[26] H. Carter, B. Mood, P. Traynor, and K. R. B. Butler. 2016. Secure outsourced garbled circuit evaluation for mobile devices. In *Journal of Computer Security*.
[27] O. Catrina and A. Saxena. 2010. Secure Computation with Fixed-Point Numbers. In *FC*.
[28] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff. 2017. Privacy-Preserving Classification on Deep Neural Network. Cryptology ePrint Archive, Report 2017/035. (2017).
[29] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. 2017. EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation. Cryptology ePrint Archive, Report 2017/1109. (2017).
[30] F. Chollet. 2015. keras. https://github.com/fchollet/keras. (2015).
[31] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. 2009. Asynchronous multiparty computation: Theory and implementation. In *PKC*.
[32] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*.
[33] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni. 2015. Automated synthesis of optimized circuits for secure computation. In *CCS*.
[34] D. Demmler, T. Schneider, and M. Zohner. 2014. Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens.. In *USENIX Security*.
[35] D. Demmler, T. Schneider, and M. Zohner. 2015. ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation.. In *NDSS*.
[36] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*.
[37] W. Du and M. J. Atallah. 2001. Protocols for secure remote database access with approximate matching. In *E-Commerce Security and Privacy*.
[38] J. Feigenbaum, B. Pinkas, R. Ryger, and F. Saint-Jean. 2004. Secure computation of surveys. In *EU Workshop on Secure Multiparty Protocols*.
[39] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *EUROCRYPT*.
[40] O. Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.
[41] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to play any mental game. In *STOC*.
[42] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. 2010. TASTY: tool for automating secure two-party computations. In *CCS*.
[43] A. Herzberg and H. Shulman. 2012. Oblivious and Fair Server-Aided Two-Party Computation. In *ARES*.
[44] E. Hesamifard, H. Takabi, and M. Ghasemi. 2017. CryptoDL: Deep Neural Networks over Encrypted Data. arXiv preprint arXiv:1711.05189. (2017).
[45] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. 2012. Secure two-party computations in ANSI C. In *CCS*.
[46] Y. Huang. 2012. *Practical Secure Two-Party Computation*. Ph.D. Dissertation. University of Virginia.
[47] Y. Huang, D. Evans, J. Katz, and L. Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits.. In *USENIX Security*.
[48] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. 2003. Extending oblivious transfers efficiently. In *CRYPTO*.
[49] W. jie Lu and J. Sakuma. 2018. Faster Multiplication Triplet Generation from Homomorphic Encryption for Practical Privacy-Preserving Machine Learning under a Narrow Bandwidth. Cryptology ePrint Archive, Report 2018/139. (2018).
[50] S. Kamara, P. Mohassel, and B. Riva. 2012. Salus: a system for server-aided secure function evaluation. In *CCS*.
[51] F. Kerschbaum, T. Schneider, and A. Schröpfer. 2014. Automatic protocol selection in secure two-party computations. In *ACNS*.
[52] V. Kolesnikov and T. Schneider. 2008. Improved garbled circuit: Free XOR gates and applications. In *ICALP*.
[53] B. Kreuter, A. Shelat, B. Mood, and K. R. Butler. 2013. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *USENIX Security*.
[54] B. Kreuter, A. Shelat, and C.-H. Shen. 2012. Billion-Gate Secure Computation with Malicious Adversaries.. In *USENIX Security*.
[55] T. Krips and J. Willemson. 2014. Hybrid model of fixed and floating point numbers in secure multiparty computations. In *ISC*.
[56] A. Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report. University of Toronto.
[57] S. Laur, H. Lipmaa, and T. Mielikäinen. 2006. Cryptographically private support vector machines. In *SIGKDD*.
[58] Y. A. Le Trieu Phong, T. Hayashi, L. Wang, and S. Moriai. 2018. Privacy-Preserving Deep Learning via Additively Homomorphic Encryption. *IEEE TIFS* (2018).
[59] Y. LeCun, C. Cortes, and C. Burges. 2017. MNIST dataset. http://yann.lecun.com/exdb/mnist/. (2017).
[60] Y. Lindell and B. Pinkas. 2000. Privacy Preserving Data Mining. In *CRYPTO*.
[61] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. 2015. ObliVM: A programming framework for secure computation. In *IEEE S&P*.
[62] J. Liu, M. Juuti, Y. Lu, and N. Asokan. 2017. Oblivious Neural Network Predictions via MiniONN transformations. In *CCS*.
[63] X. Liu, R. H. Deng, W. Ding, R. Lu, and B. Qin. 2016. Privacy-preserving outsourced calculation on floating point numbers. In *IEEE TIFS*.
[64] E. Makri, D. Rotaru, N. P. Smart, and F. Vercauteren. 2017. PICS: Private Image Classification with SVM. Cryptology ePrint Archive, Report 2017/1190. (2017).
[65] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. 2004. Fairplay-Secure Two-Party Computation System.. In *USENIX Security*.
[66] P. Mohassel, O. Orobets, and B. Riva. 2016. Efficient Server-Aided 2PC for Mobile Phones. In *PoPETs*.
[67] P. Mohassel and Y. Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.
[68] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor. 2016. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *IEEE EuroS&P*.
[69] M. Naor, B. Pinkas, and R. Sumner. 1999. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*.
[70] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. 2013. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P*.
[71] C. Orlandi, A. Piva, and M. Barni. 2007. Oblivious Neural Network Computing via Homomorphic Encryption. In *EURASIP Journal on Information Security*.
[72] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *IEEE EuroS&P*.
[73] E. Pattuk, M. Kantarcioglu, H. Ulusoy, and B. Malin. 2016. CheapSMC: A Framework to Minimize Secure Multiparty Computation Cost in the Cloud. In *DBSec*.
[74] P. Pullonen and S. Siim. 2015. Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations. In *FC*.
[75] Y. Rahulamathavan, R. C.-W. Phan, S. Veluru, K. Cumanan, and M. Rajarajan. 2014. Privacy-Preserving Multi-Class Support Vector Machine for Outsourcing the Data Classification in Cloud. In *IEEE TDSC*.
[76] M. S. Riazi, E. M. Songhori, and F. Koushanfar. 2017. PriSearch: Efficient Search on Private Data. In *DAC*.
[77] B. D. Rouhani, M. S. Riazi, and F. Koushanfar. 2018. DeepSecure: Scalable Provably-Secure Deep Learning. In *DAC*.
[78] A. Sadeghi, T. Schneider, and I. Wehrenberg. 2009. Efficient Privacy-Preserving Face Recognition. In *ICISC*.
[79] A.-R. Sadeghi and T. Schneider. 2009. Generalized Universal Circuits for Secure Evaluation of Private Functions with Application to Data Classification. In *ICISC*.
[80] T. Schneider and M. Zohner. 2013. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *FC*.
[81] R. Shokri and V. Shmatikov. 2015. Privacy-preserving deep learning. In *CCS*.
[82] S. Sivakorn, I. Polakis, and A. D. Keromytis. 2016. I am Robot: (Deep) Learning to Break Semantic Image CAPTCHAs. In *IEEE EuroS&P*.
[83] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. 2015. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *IEEE S&P*.
[84] J. Vaidya, H. Yu, and X. Jiang. 2008. Privacy-preserving SVM classification. *Knowledge and Information Systems* 14, 2 (2008), 161–178.
[85] A. Yao. 1986. How to generate and exchange secrets. In *FOCS*.
[86] S. Zahur, M. Rosulek, and D. Evans. 2015. Two Halves Make a Whole. In *EUROCRYPT*.
[87] Y. Zhang, A. Steele, and M. Blanton. 2013. PICCO: a general-purpose compiler for private distributed computation. In *CCS*.

A CIFAR-10 CNN ARCHITECTURE

In Fig. 3 we give the architecture of the CNN trained and implemented for classifying images from the CIFAR-10 dataset. The architecture is the same as the one in Fig. 13 in [62]. We also list the protocols used to execute each layer of the CNN in Chameleon and the necessary protocol conversions.

Layer	Description	Protocol
Convolution	Input image $3 \times 32 \times 32$, window size 3×3, stride $(1,1)$, pad $(1,1)$, number of output channels 64: $\mathbb{R}^{64\times1024} \leftarrow \mathbb{R}^{64\times27} \cdot \mathbb{R}^{27\times1024}$.	A-SS
	A2GMW	
ReLu Activation	Computes ReLu for each input.	GMW
	GMW2A	
Convolution	Window size 3×3, stride $(1,1)$, pad $(1,1)$, number of output channels 64: $\mathbb{R}^{64\times1024} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times1024}$.	A-SS
	A2GMW	
ReLu Activation	Computes ReLu for each input.	GMW
	GMW2A	
Mean Pooling	Window size $1 \times 2 \times 2$, outputs $\mathbb{R}^{64\times16\times16}$.	A-SS
Convolution	Window size 3×3, stride $(1,1)$, pad $(1,1)$, number of output channels 64: $\mathbb{R}^{64\times256} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times256}$.	A-SS
	A2GMW	
ReLu Activation	Computes ReLu for each input.	GMW
	GMW2A	
Convolution	Window size 3×3, stride $(1,1)$, pad $(1,1)$, number of output channels 64: $\mathbb{R}^{64\times256} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times256}$.	A-SS
	A2GMW	
ReLu Activation	Computes ReLu for each input.	GMW
	GMW2A	
Mean Pooling	Window size $1 \times 2 \times 2$, outputs $\mathbb{R}^{64\times16\times16}$.	A-SS
Convolution	Window size 3×3, stride $(1,1)$, pad $(1,1)$, number of output channels 64: $\mathbb{R}^{64\times64} \leftarrow \mathbb{R}^{64\times576} \cdot \mathbb{R}^{576\times64}$.	A-SS
	A2GMW	
ReLu Activation	Computes ReLu for each input.	GMW
	GMW2A	
Convolution	Window size 1×1, stride $(1,1)$, number of output channels 64: $\mathbb{R}^{64\times64} \leftarrow \mathbb{R}^{64\times64} \cdot \mathbb{R}^{64\times64}$.	A-SS
	A2GMW	
ReLu Activation	Computes ReLu for each input.	GMW
	GMW2A	
Convolution	Window size 1×1, stride $(1,1)$, number of output channels 16: $\mathbb{R}^{16\times64} \leftarrow \mathbb{R}^{16\times64} \cdot \mathbb{R}^{64\times64}$.	A-SS
	A2GMW	
ReLu Activation	Computes ReLu for each input.	GMW
	GMW2A	
Fully Connected Layer	Fully connects the incoming 1024 nodes to the outgoing 10 nodes: $\mathbb{R}^{10\times1} \leftarrow \mathbb{R}^{10\times1024} \cdot \mathbb{R}^{1024\times1}$.	A-SS
	A2GC	
Arg Max	Extracts the label of the class with the highest probability.	GC

Figure 3: The architecture of the CNN trained from the CIFAR-10 dataset (taken from [62]) and the protocols used to execute each layer in Chameleon, including the necessary protocol conversions.

B SUPPORT VECTOR MACHINES (SVMS)

One of the most frequently used classification tools in machine learning and data mining is the Support Vector Machine (SVM). An SVM is a supervised learning method in which the model is created based on labeled training data. The result of the training phase is a non-probabilistic *binary* classifier. The model can then be used to classify input data $\mathbf{x}$ which is a d-dimensional vector. In Chameleon, we are interested in a scenario where the server holds an already trained SVM model and the user holds the query $\mathbf{x}$. Our goal is to classify the user's query without disclosing the user's input to the server or the server's model to the user.

The training data, composed of N d-dimensional vectors, can be viewed as N points in a d-dimensional space. Each point i is labeled as either $\mathbf{y_i} \in \{-1, 1\}$, indicating which class the data point belongs to. If the two classes are linearly separable, a $(d-1)$-dimensional hyperplane which separates these two classes can be used to classify future queries. A new query point can be labeled based on which side of the hyperplane it resides on. The hyperplane is called *decision boundary*. While there can be infinitely many such hyperplanes, a hyperplane is chosen that maximizes the margin between the two classes. That is, a hyperplane is chosen such that the distance between the nearest point of each class to the hyperplane is maximized. Those training points that reside on the margin are called *support vectors*. This hyperplane is chosen to achieve the highest classification accuracy. Fig. 4 illustrates an example in two-dimensional space. The optimal hyperplane can be represented using a vector $\mathbf{w}$ and a distance from the origin $\mathbf{b}$. Therefore, the optimization task can be formulated as:

$$\text{minimize } \|\mathbf{w}\| \text{ s.t. } \mathbf{y_i}(\mathbf{w} \cdot \mathbf{x_i} - \mathbf{b}) \geqslant 1,\ i = 1, 2, \ldots, N$$

The size of the margin equals $\mathbf{M} = \frac{2}{\|\mathbf{w}\|}$. This approach is called hard-margin SVM.

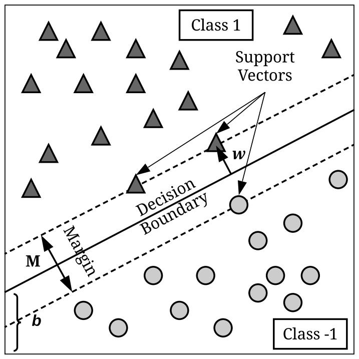

Figure 4: Classification using Support Vector Machine (SVM).

An extension of the hard-margin SVM, called a soft-margin SVM, is used for scenarios where the two classes are not linearly separable. In this case, the hinge lost function is used to penalize if the training sample is residing on the wrong side of the classification boundary. As a result, the optimization task is modified to:

$$\frac{1}{N} \sum_{i=1}^{N} \max\,(0, 1 - \mathbf{y_i}(\mathbf{w} \cdot \mathbf{x_i} - \mathbf{b})) + \lambda\|\mathbf{w}\|^2$$

where λ is a parameter for the tradeoff between the size of the margin and the number of points that lie on the correct side of the boundary.

For both soft-margin and hard-margin SVMs, the performed classification task is similar. The output label of the user's query is computed as:

$$\text{label} \in \{-1, 1\} = \text{sign}(\mathbf{w} \cdot \mathbf{x} - \mathbf{b})$$

We run our experiments using the same setup described in §6. The results of the experiments are provided in Tab. 6 for feature vector sizes of 10, 100, and 1,000.

Table 6: Classification time (in seconds) and communication costs (in kilobytes) of Chameleon using SVM models for different feature sizes in the LAN setting.

	Classification Time (ms)			Communication (kB)		
Feature Size	Offline	Online	Total	Offline	Online	Total
10	8.91	0.97	9.88	3.2	3.3	6.5
100	9.49	0.99	10.48	3.9	4.7	8.7
1000	10.28	1.14	11.42	11.1	19.1	30.3

Comparison with Previous Works. Makri et al. [64] present PICS, a private image classification system based on SVM learning. They evaluate their implementation in SPDZ [32] with two computation nodes. For one binary classification with 20 features, they report 145 s/30 ms offline/online run-time. Although in a different security and computational model, Chameleon performs the same task four orders of magnitude faster. Bos et al. [20] study privacy-preserving classification based on hyperplane decision, Naive Bayes, and decision trees using homomorphic encryption. For a credit approval dataset with 47 features, they report a run-time of 217 ms and 40 kB of communication, whereas, Chameleon can securely classify a query with 1,000 features in only 11.42 ms with 30.3 kB of communication. Rahulamathavan et al. [75] also design a solution based on homomorphic encryption for binary as well as multi-class classification based on SVMs. In the case of binary classification, for a dataset with 9 features, they report 7.71 s execution time and 1.4 MB communication. In contrast, for the same task, Chameleon requires less than 10 ms execution time and 6.5 kB of communication. Laur et al. [57] provide privacy-preserving training algorithms based on general kernel methods. They also study privacy-preserving classification based on SVMs but they do not report any benchmark results. Vaidya et al. [84] propose a method to train an SVM model where the training data is distributed among multiple parties. This scenario differs from ours where we are interested in the SVM-based classification. As a proof-of-concept, we have focused on SVM models for linear decision boundaries. However, Chameleon can be used for non-linear decision boundaries as well.

C BENCHMARKS OF ATOMIC OPERATIONS

We benchmark different atomic operations in Chameleon and compare them with three prior art frameworks: TinyGarble [83], ABY [35], and Sharemind [18]. The result for ABY is reported for three different scenarios: GC-only, GMW-only, and Additive SS-only. For TinyGarble, ABY, and Chameleon we run the frameworks ourselves. The benchmarking environment remains the same as described in §6. Unlike TinyGarble and ABY, Sharemind lacks built-in atomic benchmarks and is a commercial product that requires contracting even for academic purposes. Thus, we give the results from the original paper [18] and justify why Chameleon performs better on equal hardware.

We do not include WAN benchmarks of atomic operations for the following reason: Due to higher latency, GC-based circuit evaluation with constant rounds is preferred instead of GMW for binary operations. However, since the atomic benchmarks do not measure input sharing (for which GC uses STP-aided OT generation), no difference is visible to prior art.

Evaluation Results. The detailed run-times and communication costs for arithmetic and binary operations are given in Tab. 7 and in Tab. 8, respectively. The highlighted area for ABY-A in both tables reflects that ABY does not perform these operations in additive secret sharing. The highlighted area in Tab. 8 for Sharemind indicates that the corresponding information is not reported in the original paper. Tab. 9 additionally shows the run-times for conversions between different sharings.[5] All reported run-times are the average of 10 executions with less than 15 % variance.

As can be seen, Chameleon outperforms all state-of-the-art frameworks. Run-times and communication for arithmetic operations in Chameleon are only given in A-SS since from the ABY results and Tab. 9 it follows that even for a single addition or multiplication operation it is worthwhile to perform a protocol conversion. The remaining atomic operations for Chameleon are given in Boolean sharing where we observe major improvements over ABY due to our efficient B-MT precomputation.[6] Regarding conversion operations, the GMW2A, GMW2GC, and A2GC performance in Chameleon benefits from fast STP-aided OTs (cf. §5.5).

Although, the experimental setup of Sharemind is computationally weaker than ours, we emphasize that Chameleon is more efficient because of the following reasons: (i) To compute each MULT operation, the Sharemind version benchmarked in [18] requires 6 instances of the Du-Atallah protocol whereas our framework needs only 2. (ii) In Sharemind, bit-level operations such as XOR/AND require a bit-extraction protocol, which is computationally expensive. Please note that these costs are not reported in [18] and hence are not reflected in Tab. 7. (iii) Operations such as CMP, EQ, and MUX can most efficiently be realized using GC/GMW protocols and as a result, Chameleon can perform these operations faster.

The run-times for TinyGarble include base OTs, online OTs, garbling/evaluating, and data transmission. This is why the run-time for MULT is not significantly higher than for other operations that require orders of magnitude fewer gates. However, in Chameleon, we precompute all OTs, which significantly reduces the run-time.

Note that the shown run-times and communication results for Chameleon represent the worst case, namely for the party that receives additional data from the STP besides the required seeds for OT and MT generation.[7]

Communication in the Offline Phase. The communication costs of the offline phase in Chameleon are compared to ABY [35] in Tab. 10. To generate a single B-MT, Chameleon requires only a constant-size data transmission to one party and 256× less communication to the other party compared to ABY. When generating a single A-MT, the required communication to the other party is reduced by factor 273×/289×/321× for a bitlength of 16/32/64, respectively. This is a significant enhancement since in most machine learning applications, the main bottleneck is the vector/matrix multiplication, which requires a large amount of A-MTs.

[5] The required *communication* for conversion operations equals ABY [35] since STP-aided OT generation does not reduce the amount of communication (cf. Tab. 10).

[6] The benchmarking methodology inherited from ABY omits input sharing, which is why no improvement for GC-based operations is measurable compared to ABY.

[7] An improved implementation could equally distribute computation and communication among the two parties by dividing the data sent by the STP evenly, thereby further reducing the run-times.

Table 7: Run-Times (in milliseconds unless stated otherwise) for different atomic operations and comparison with prior art. Each experiment is performed for 1,000 operations on 32-bit numbers in parallel. The detailed performance results for ABY [35] are provided for three different modes of operation: GC, GMW, and Additive. Minimum values marked in bold.

	TinyGarble [83]	**ABY-GC** [35]		**ABY-GMW** [35]		**ABY-A** [35]		**Sharemind** [18]	**Chameleon**	
Op	Online	Offline	Online	Offline	Online	Offline	Online	Online	Offline	Online
ADD	1.57 s	11.71	2.73	25.78	4.73	**0.00**	**0.00**	1 μs	**0.00**	**0.00**
MULT	2.31 s	423.82	112.29	174.52	14.25	10.46	0.59	17	**4.24**	**0.13**
XOR	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**			1 μs	**0.00**	**0.00**
AND	1.58 s	11.83	2.34	9.27	**0.52**			17	**1.50**	0.56
CMP	1.57 s	11.90	2.63	17.39	1.63			2.5 s	**2.46**	**1.48**
EQ	1.56 s	11.60	2.42	9.11	1.15			5 s	**1.54**	**1.09**
MUX	1.59 s	11.91	2.49	**1.06**	0.68			34	1.52	**0.63**

Table 8: Communication (in kilobytes unless stated otherwise) for different atomic operations and comparison with prior art. Each experiment is performed for 1,000 operations on 32-bit numbers in parallel. The detailed performance results of the ABY framework [35] is provided for three modes of operation: GC, GMW, and Additive. Minimum values marked in bold.

	TinyGarble [83]	**ABY-GC** [35]		**ABY-GMW** [35]		**ABY-A** [35]		**Sharemind** [18]	**Chameleon**	
Op	Total	Offline	Online	Offline	Online	Offline	Online	Total	Offline	Online
ADD	7936	992	**0**	3593	76	**0**	**0**	**0**	**0**	**0**
MULT	318 K	47649	**0**	37900	840	1280	16	192	**8**	16
XOR	**0**	**0**	**0**	**0**	**0**			**0**	**0**	**0**
AND	8192	1024	**0**	1028	16			192	**12**	8
CMP	8192	1024	**0**	2851	45				**23**	33
EQ	7936	992	**0**	995	16				**8**	12
MUX	8192	1024	**0**	33	8			384	**8**	4

Table 9: Run-Times (in milliseconds) for conversion operations and comparison with prior art. Each experiment is performed for 1,000 operations on 32-bit numbers in parallel. Minimum values marked in bold.

	ABY [35]		**Chameleon**	
Op	Offline	Online	Offline	Online
GC2GMW	**0.00**	**0.00**	**0.00**	**0.00**
GMW2A	9.47	2.44	**3.45**	**2.33**
GMW2GC	17.05	1.30	**13.24**	**1.15**
A2GC	19.75	14.03	**15.83**	**12.91**

Table 10: Communication (in bits) in the offline phase in Chameleon compared to prior art ABY [35].

	ABY [35]	**Chameleon**	**Improvement**
OT	128	128	-
B-MT	256	1	256×
A-MT (bitlength $\ell = 16$)	4,368	16	273×
A-MT (bitlength $\ell = 32$)	9,248	32	289×
A-MT (bitlength $\ell = 64$)	20,544	64	321×

D FURTHER RELATED WORKS ON PRIVACY-PRESERVING MACHINE LEARNING

One of the earliest solutions for obliviously evaluating a neural network was proposed by Orlandi et al. [71]. They suggest adding fake neurons to the hidden layers in the original network and evaluating the network using HE. Chabanne et al. [28] also approximate the ReLu non-linear activation function using low-degree polynomials and provide a normalization layer prior to the activation layer. However, they do not report experimental results. Sadeghi and Schneider proposed to utilize universal circuits to securely evaluate neural networks and fully hide their structure [79]. Privacy-preserving classification of electrocardiogram (ECG) signals using neural networks has been addressed in [10]. The recent work of Shokri and Shmatikov [81] is a Differential Privacy (DP) based approach for the distributed training of a Neural Network and they do not provide secure DNN or CNN inference. Due to the added noise in DP, any attempt to implement secure inference suffers from a significant reduction in accuracy of the prediction. Phong et al. [58] propose a mechanism for privacy-preserving deep learning based on additively homomorphic encryption. They do not consider secure deep learning inference (classification). There are also limitations of deep learning when an adversary can craft malicious inputs in the training phase [72]. Moreover, deep learning can be used to break semantic image CAPTCHAs [82].