PrivPy: General and Scalable Privacy-Preserving Data Mining

Yi Li
Tsinghua University
Beijing, China
xiaolixiaoyi@gmail.com

Wei Xu
Tsinghua University
Beijing, China
wei.xu.0@gmail.com

ABSTRACT

Privacy is a big hurdle for collaborative data mining across multiple parties. We present multi-party computation (MPC) framework designed for large-scale data mining tasks. PrivPy combines an easy-to-use and highly flexible Python programming interface with state-of-the-art secret-sharing-based MPC backend. With essential data types and operations (such as NumPy arrays and broadcasting), as well as automatic code-rewriting, programmers can write modern data mining algorithms conveniently in familiar Python. We demonstrate that we can support many real-world machine learning algorithms (e.g. logistic regression and convolutional neural networks) and large datasets (e.g. 5000-by-1-million matrix) with minimal algorithm porting effort.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**; • **Computing methodologies → Cooperation and coordination**.

KEYWORDS

privacy-preserving, data mining, numpy, python

ACM Reference Format:
Yi Li and Wei Xu. 2019. PrivPy: General and Scalable Privacy-Preserving Data Mining. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3292500.3330920

1 INTRODUCTION

Privacy is an important issue in big data age. The success of data mining is often built on data, and it is often desirable to integrate data from multiple sources for better mining results. However, the unrestricted exchanging of sensitive data may threaten users' privacy and is often prohibited by laws or business practices. How to protect privacy while allowing the integration of multiple data sources demands prompt solutions.

Secure multi-party computation (MPC) allows players to collectively compute a function without revealing private information except for the final output. MPC often uses various cryptographic primitives, such as garbled circuit [48] and secret sharing [42], with different efficiency and security assumptions. After more than 30 years of development, we have started to see that the real-world data mining applications start to use MPC [29, 47]. However, numerous challenges still exist that prevent widespread adoption of secure computation techniques.

One of the most important issue hindering MPC's adoption is programmability, especially for "big data" applications. Despite of the development of efficiency improvement of MPC during the past decades, existing MPC solutions often ignore the core requirements of data mining applications. They either require considerable expertise in cryptography to understand the cost of each operation, or use special programming languages with high learning curves [6, 14, 20, 30, 35, 41]. Some useful solutions, such as [45], though providing rich interfaces for MPC, mainly focus on basic MPC operations, including not only basic arithmetics but also low-level cryptography tools such as *oblivious transfer* [40]. In contrast, machine learning programmers use Python-based frameworks like PyTorch [37], Tensorflow [1] and Scikit-learn [38] with built-in support of high-level data types like real numbers, vectors and matrices, as well as non-linear functions such as the logistic function and ReLu. It is almost impossible for data scientists to rebuild and optimize all these often taken-for-granted primitives in a modern machine learning package in an MPC language. On the other hand, it is also costly for MPC experts to rewrite all the machine learning algorithm packages. Thus, it is essential to design an MPC front-end that is friendly with the data mining community, which is Python with NumPy [44] nowadays. Actually, many machine learning frameworks use Python front-ends and provide Numpy-style array operations to ease machine learning programming.

In this paper, we propose PrivPy, an efficient framework for privacy-preserving collaborative data mining, aiming to provide an elegant end-to-end solution for data mining programming. The PrivPy front-end provides Python interfaces that resemble those from NumPy, one of the most popular Python packages, as well as a wide range of functions commonly used in machine learning. We also provide an computation engine which is based on secret sharing and provides efficient arithmetics. We would like to stress that the main goal of PrivPy is not to make theoretic breakthrough in cryptographic protocols, but rather to build a practical system that enables elegant machine learning programming on secure computation frameworks and makes right trade-offs between efficiency and security. In particular, we make the following contributions:

(1) **Python programming interface with high-level data types.** We provide a very clean Python language integration with privacy-enabled common operations and high-level primitives,

Due to the first author's negligence and technical difficulties, we accidentally left out the following coauthors who made significant contributions to this article. They are: 1. Yitao Duan, NetEase Youdao, duan@rd.netease.com; 2. Shuoyao Zhao, Shanghai Jiaotong University, zhao_sy2016@sjtu.edu.cn; 3. Yu Yu, Shanghai Jiaotong University, yuyuathk@gmail.com.

KDD '19, August 4–8, 2019, Anchorage, AK, USA

ACM ISBN 978-1-4503-6201-6/19/08…$15.00
https://doi.org/10.1145/3292500.3330920

including *broadcasting* that manipulates arrays of different shapes, and the *ndarray* methods, two Numpy [44] features widely utilized to implement machine learning algorithms, with which developers can port complex machine learning algorithms onto PrivPy with minimal effort.

(2) **Automatic code check and optimization.** Our front-end will help the programmers avoid "performance pitfalls", by checking the code and optimizing it automatically.

(3) **Decoupling programming front-end with computation back-ends.** We introduce a general private operator layer to allow the same interface to support multiple computation back-ends, allowing trade-offs among different performance and security assumptions. Our current implementation supports the SPDZ back-end, the ABY3 back-end and our own computation engine.

(4) **Validation on large-scale machine learning tasks.** We demonstrate the practicality of our system for data mining applications, such as data query, logistic regression and convolutional neural network (CNN), on real-world datasets and application scenarios. We also show that our system can scale to large-scale tasks by transparently manipulating a 5000-by-1-million matrix.

2 RELATED WORK

In this paper, we mainly focus on privacy-preserving computation systems for general arithmetics, especially for data mining tasks. A practical such system includes two parts: an efficient computation engine and an easy-to-use programming front-end.

Frameworks based on (fully) homomorphic encryption [18, 32] are impractical due to heavy computation overhead. Approaches based on garbled circuit (GC) [4, 34, 45, 50] will be impractical for general-purpose arithmetical computations, especially for various kinds of machine learning algorithms, as they are costly in bandwidth. There are also many MPC frameworks using secret sharing supporting general arithmetics. For example, [3] performs integer/bit multiplication with 1 round and optimal communication cost using three semi-honest servers. SPDZ [14] uses addition secret sharing and can tolerate up to $n-1$ corrupted parties. While natively supporting efficient integer operations, most of them (e.g. [7, 14]) support real numbers by parsing each shared integer into m field elements (m is the bit length of the each field element) and use bit-level operations to simulate fixed/floating point operations [11, 24, 26], thus requires each party to send $O(m)$ meessages. SecureML [35] is based on two-party secret sharing and provides built-in fixed-point multiplication with $O(1)$ message complexity, but requires expensive precomputation to generate Beaver multiplication triples. ObliviousNN [31] optimizes the performance of dot product, but suffers similar problem with SecureML. ABY3 [33], which extends the work of [3] and provides three-party computation, is the state-of-the-art for general arithmetics. To perform fixed-point multiplication, ABY3 provides two alternatives: one requires a lightweight precomputation and each party needs to send no more than 2 messages in 1 round in the online phase, while the other requires no precomputation and each party sends no more than 2 messages, but needs 2 rounds. In comparison, our computation engine, which also provides built-in support for fixed-points, performs fixed-point multiplication in 1 round without precomputation and each party only sends 2 messages.

We emphasize that the adoption of privacy-preserving computation is beyond the computation efficiency and the programmability is as the same importance. TASTY [20] and ABY [16] provide interfaces for programmers to convert between different schemes. However, they only expose low-level interfaces and the programmers should decide by themselves which cryptographic tools to choose and when to convert them, making the learning curve steep. L1 [41] is an intermediate language for MPC and supports basic operations. But L1 is a domain-specific language and does not provide high-level primitives to ease array/matrix operations frequently used in machine learning algorithms. [15] and [9] suffer from similar problems. PICCO [51] supports additive secret sharing and provides customized C-like interfaces. But the interfaces are not intuitive enough and only support simple operations for array. Also, according to their report, the performance is not practical enough for large-scale arithmetical tasks. KSS [25] and ObliVM [30] also suffer from these issues. [13] provides a compiler for SPDZ and [2] extends it to support more MPC protocols. But they are still domain-specific and do not provide enough high-level primitives for machine learning tasks. PrivPy, on the other hand, stays compatible with Python and provides high-level primitives (e.g. broadcasting) with automatic code check and optimization, requiring no learning curve on the application programmer side, making it possible to implement machine learning algorithms conveniently in a privacy-preserving situation.

3 PRIVPY DESIGN OVERVIEW

3.1 Problem formulation

Application scenarios. We identify the following two major application scenarios for privacy-preserving data mining:

- **multi-source data mining.** It is common that multiple organizations (e.g. hospitals), each independently collecting part of a dataset (e.g. patients' information), want to jointly train a model (e.g. for inferring a disease), without revealing any information.
- **inference with secret model and data.** Sometimes the parameters of a model are valuable. For example, the credit scoring parameters are often kept secret. Neither the model owner nor the data owner want to leak their data in the computation.

Problem formulation. We formulate both scenarios as an MPC problem: there are n clients $C_i (i = 1, 2, \ldots, n)$. Each C_i has a set of private data D_i as its input. The goal is to use the union of all D_i's to compute some function $o = f(D_1, D_2, \ldots, D_n)$, while no private information other than the output o is revealed during the computation. D_i can be records collected independently by C_i, and C_i's can use them to jointly train a model or perform data queries.

Security assumptions. Our design is based on two widely adopted assumptions in the security community [3, 7, 29]: 1) All of the servers are *semi-honest*, which means all servers follow the protocol and would not conspire with other servers, but they are curious about the users' privacy and would steal information as much as possible; and 2) all communication channels are secure and adversaries cannot see/modify anything in these channels. In practice, as there is a growing number of independent and competing cloud providers, it is feasible to find a small number of such servers. We leave extensions of detecting malicious adversaries as future work.

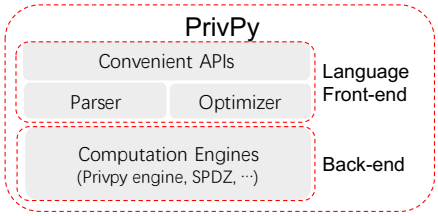

Figure 1: The overview of PrivPy architecture.

3.2 Design overview

Fig. 1 shows an overview of PrivPy design, which has two main components: the *language front-end* and the *computation engine back-end*. The front-end provides programming interfaces and code optimizations. The back-end performs the secret-sharing-based privacy-preserving computation. We discuss our key design rationals in this section.

Decoupling the frontend with backend. We decouple the front-end and back-end using an extensible interface. The major benefit is that we can adapt to multiple language and backend techniques. While we believe Python is a natural choice for the frontend for the data mining community, we support multiple MPC backends to allow tradeoffs among different security assumptions and performance. Our interface between frontend and backend is extensible. The basic interface only require scaler data types and operations, making it possible to use very simple engines. Then we add extension interfaces to fully utilize backends with performance-optimizations such as array types and complex computation functions (e.g. vector outer product). It is an analogous to the extensible instruction set architecture (ISA) design. In fact, we currently support three backends: our own backend, SPDZ [14], ABY3 [33].

Focus on performance optimizations for the entire algorithm. Performance is the key to enable scalable data mining tasks. We optimize performance at three different levels: 1) optimize single operation performance using the 2-out-of-4 secret sharing protocols; 2) batch up operations whenever possible; 3) perform language-level optimizations in the language frontend.

Based on 2-out-of-4 secret sharing protocol. Handling collaborative data mining tasks over multiple parties has some two unique performance challenge: 1) the computation might happen on wide area networks (WANs), and thus it is both bandwidth and latency sensitive; and 2) there are vast amount of data, and thus we need to minimize the overall computation, including the preprocessing. Existing engines either require multiple rounds of communication and thus perform poorly in WANs ([7, 14]), or require significant amount of pre-computation ([33, 35]).

We design a 2-out-of-4 secret sharing protocol combining the ideas in SecureML [35] and ABY3 [33]. By adding a fourth server, we can eliminate the pre-computation in ABY3, but keep its one-round only online communication feature for fixed-point multiplication while preserving the same online communication complexity. Also, the correctness proof of our protocol directly follows [33] and [35], making it simple to establish the correctness.

Hierarchical private operations (POs). We call operations on private variable *private operations (POs)*. Table 1 provides an overview of different POs implemented in PrivPy.

type	operations			
basic	add/sub	multiplication	oblivious transfer	bit extraction
derived	comparison	sigmoid	relu	division
	log	exp	sqrt	abs
ndarray	all	any	append	argmax
	argmin	argparition	argsort	clip
	compress	copy	cumprod	cumsum
	diag	dot	fill	flatten
	item	itemset	max	mean
	min	ones	outer	partition
	prod	ptp	put	ravel
	repeat	reshape	resize	searchsorted
	sort	squeeze	std	sum
	swapaxes	take	tile	trace
	transpose	var	zeros	

Table 1: Supported operations of the PrivPy front-end.

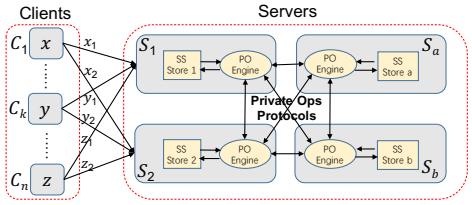

Figure 2: The overview of PrivPy computation engine.

We identified a number of POs that are either essential for computation or performance critical, and we implement them directly in secret sharing. We call them basic POs. Limited by space, we only introduce the fixed-point number multiplication PO in Section 4.2. Another set of POs we implement is to support vector and matrix operations. It is essential to support the array types in Python.

One good feature of the 2-out-of-4 secret sharing is that the result of the computation is still secret shares with exactly the same format. Thus, we can concatenate different POs together and implement derived POs. Note that even derived POs performs better than a Python-library as it is pre-compiled in the engine - much like the built-in routines in a database system.

Using frontend to provide both flexibility and performance. PrivPy frontend not only makes it easy for data scientists to write in familiar Python, but also it provide extensive optimizations to support array types, including arbitrary sized arrays and operations. Also, it automatically performs code rewriting to help programmers to avoid common performance pit-falls.

Engine architecture. We use four (semi-honest) servers to implement the 2-out-of-4 secret sharing protocol above in the PrivPy engine: $\mathbf{S_1}$, $\mathbf{S_2}$, $\mathbf{S_a}$ and $\mathbf{S_b}$. We adopt a client/server model, just like many existing MPC systems [3, 7, 17, 29, 30]. Clients send secretly shared data to the servers, then the servers perform privacy-preserving computation on these shares (see Fig. 2).

Each of the four servers has two subsystems. The *secret sharing storage (SS store) subsystem* provides (temporary) storage of shares of private inputs and intermediate results. while the *private operation (PO) subsystem* provides an execution environment for private operations. The servers read shares from the SS store, execute a PO, and write the shares of the result back to the SS store. Thus we can compose multiple POs to form a larger PO or a complex algorithm.

Task execution. In summary, PrivPy runs a machine learning tasks in the following four steps: 1) the Python front-end analyzes

and rewrites the program for algorithm-level performance optimization. 2) Each client computes the secret shares for her private variables, and sends the resulting shares to the servers. 3) All servers runs the Python code in parallel on the private shares without any client involvement, until it reached the `reveal()` point in the code. 4) The servers invoke `reveal()`, the clients are notified to find the `result` shares, and finally recover the cleartext result.

4 THE PRIVPY COMPUTATION ENGINE

In this section, using multiplication as an example, we introduce our secret sharing protocol design, and then we describe the private operation (POs) we support. The thread model (security assumptions) is defined in Section 3.1.

4.1 Replicated 2-out-of-4 secret sharing

Secret Sharing. Secret sharing encodes a secret number into multiple shares and distributes the shares to a group of participants such that no information about the raw number is revealed as long as no sufficient is gathered. The simplest secret sharing is to encode a number x into two numbers r and $x - r$ where r is a random number. Thus one can reconstruct x only if he gets both shares.

Replicated 2-out-of-4 secret sharing. We combine the thought of SecureML [35] and ABY3 [33], and design a more efficient protocol for fixed-point multiplication.

We define a 2-out-of-4 secret sharing, denoted as $\binom{4}{2}$-sharing, to enable efficient arithmetics. And we consider all the shares are in a $\mathbb{Z}_{2^n}$ ring. Concretely, to share an integer $x \mod 2^n$, we encode it as a tuple of shares: $[\![x]\!] = (x_1, x_1', x_2, x_2', x_a, x_a', x_b, x_b')$. $\mathrm{S_1}$ holds (x_1, x_1') where x_1 and x_1' are two independent random integers; $\mathrm{S_2}$ holds (x_2, x_2') where $x_2 = x - x_1$ and $x_2' = x - x_1'$; $\mathrm{S_a}$ holds (x_a, x_a') where $x_a = x_2$ and $x_a' = x_1'$; $\mathrm{S_b}$ holds (x_b, x_b') where $x_b = x_1$ and $x_b' = x_2'$. It can be easily seen that the two numbers each server holds are independently uniformlly distributed in the ring $\mathbb{Z}_{2^n}$, thus the non-colluding servers learns nothing about x. Note that all the linear arithmetic operations of the secret shares, such as $+, -, \times$, are over the ring $\mathbb{Z}_{2^n}$. The division operation $x/2^d$ stands for shifting the bits of x right in the two's-complement representation.

Sharing initialization. To share a number x, a client encodes it to x_1 and x_2 where x_1 is randomly sampled in $\mathbb{Z}_{2^n}$ and $x_2 = x - x_1$. Then it sends x_1 and x_2 to $\mathbf{S_1}$ and $\mathbf{S_2}$ respectively. After receiving the shares, $\mathbf{S_1}$ sends x_1 to $\mathbf{S_b}$, and $\mathbf{S_2}$ sends x_2 to $\mathbf{S_a}$. Thus the four servers holds x_1, x_2, x_a, x_b respectively. Meanwhile, $\mathbf{S_1}$ and $\mathbf{S_2}$ generate a random number r using the same seed, and calculate $x_1' = x_1 - r$ and $x_2' = x_2 + r$ respectively. Finally, $\mathbf{S_1}$ sends x_1' to $\mathbf{S_a}$ while $\mathbf{S_2}$ sends x_2' to $\mathbf{S_b}$, and the servers get the $\binom{4}{2}$-sharing of x. In this situation, each server only sees two independent random integers that are uniformly distributed in $\mathbb{Z}_{2^n}$, and no information about x is revealed to each server.

Free addition. It can be easily seen that the $\binom{4}{2}$-sharing over $\mathbb{Z}_{2^n}$ is additively homomorphic, i.e., $[\![x]\!] + [\![y]\!] = [\![x + y]\!]$, and the result shares still satisfy the above constraints. Thus each server can locally compute the share of the sum. Similarly, for shared bits over $\mathbb{Z}_2$, the `XOR` operation is also free.

Supporting decimals. We map a decimal x to $\mathbb{Z}_{2^n}$ as a fixed-point number: we scale it by a factor of 2^d and represent the rounded integer $\widetilde{x} = \lfloor 2^d x \rfloor$ as a n-bit integer using the two's-complement reprensentation. This mapping scheme is commonly used (see [10, 11, 33, 35], and the precision is 2^{-d}. It is obvious that, while ignoring the precision loss, $[\![\widetilde{x}]\!]$ remains additively homomorphic.

Protocol 1: Fixed-point multiplication protocol.

Input: Shares of two fixed-point values $[\![x]\!]$ and $[\![y]\!]$
Output: $[\![z]\!]$ where $z = xy/2^d$
Steps:

a) $\mathrm{S_1}$ generates r_{12} and r_{12}' and calculates $t_1 = x_1 y_1' - r_{12}$ and $t_1' = x_1' y_1 - r_{12}'$. Then it sends t_1 to $\mathrm{S_b}$ and sends t_1' to $\mathrm{S_a}$.
b) $\mathrm{S_2}$ generates r_{12} and r_{12}' and calculates $t_2 = x_2 y_2' + r_{12}$ and $t_2' = x_2' y_2 + r_{12}'$. Then it sends t_2 to $\mathrm{S_a}$ and sends t_2' to $\mathrm{S_b}$.
c) $\mathrm{S_a}$ generates r_{ab} and r_{ab}' and calculates $t_a = x_a y_a' - r_{ab}$ and $t_a' = x_a' y_a - r_{ab}'$. Then it sends t_a to $\mathrm{S_2}$ and sends t_a' to $\mathrm{S_1}$.
d) $\mathrm{S_b}$ generates r_{12} and r_{12}' and calculates $t_b = x_b y_b' + r_{ab}$ and $t_b' = x_b' y_b + r_{ab}'$. Then it sends t_b to $\mathrm{S_1}$ and sends t_b' to $\mathrm{S_2}$.
e) $\mathrm{S_1}$ sets $z_1 = (t_1 + t_b)/2^d$ and $z_1' = (t_1' + t_a')/2^d$;
$\mathrm{S_2}$ sets $z_2 = (t_2 + t_a)/2^d$ and $z_2' = (t_2' + t_b')/2^d$;
$\mathrm{S_a}$ sets $z_a = (t_a + t_2)/2^d$ and $z_a' = (t_a' + t_1')/2^d$;
$\mathrm{S_b}$ sets $z_b = (t_b + t_1)/2^d$ and $z_b' = (t_b' + t_2')/2^d$.

4.2 Fixed-point multiplication.

To support efficient fixed-point multiplication, we combine the thought of SecureML [35] and ABY3 [33] which are also secret-sharing based approaches. But compared with SecureML and ABY3, our fixed-point multiplication does not require precomputation and only needs 1 round of communication, and preserves the same online communication complexity for each server when ultilizing *fully-duplex* communication, as Protocol 1 shows. A remarkable thing is that, each pair of servers share a random string and use the string as the seed of a pseudorandom function, like [3, 33, 34] do, thus they can get a same random number without communication. The thought of this protocol is from [3, 33, 35] and the security is similar with them. Thus we omit the proof detail here due to space limitation. To argue the correctness, we observe that

$$\begin{aligned}
&2^d z_1 + 2^d z_2 \\
&= (x_1 y_1' - r_{12} + x_b y_b' + r_{ab}) + (x_2 y_2' + r_{12} + x_a y_a' - r_{ab}) \\
&= x_1 y_1' + x_b y_b' + x_2 y_2' + x_2 y_a' \\
&= x_1 y_1' + x_1 y_2' + x_2 y_2' + x_2 y_1' \\
&= (x_1 + x_2)(y_1' + y_2') = xy
\end{aligned}$$

This means that $(2^d z_1, 2^d z_2)$, namely $(t_1 + t_b, t_2 + t_a)$, is a two-party share of xy. Thus, according to the theorem in [35], (z_1, z_2) is a two-party share of $xy/2^d$ i.e. $z_1 + z_2 = xy/2^d$ with extremely high probability. The same applies to z_1' and z_2', i.e. $z_1' + z_2' = xy/2^d$. Also, we can see that $z_1 = z_b$, $z_2 = z_a$, $z_1' = z_a'$ and $z_2' = z_b'$. This means that the result shares of z still satisfy the constraint of $\binom{4}{2}$-sharing.

4.3 Other basic POs

Limited by space, we only briefly introduce other POs here. We implement the *comparison* operation using oblivious transfer which enables secure selection between two private numbers and bit extraction which extracts a specific bit of a private number, in a similar way as [33], and thus the correctness and security directly follow [33]. We also implement two basic *bitwise operations* XOR and AND, and we can get all kinds of bitwise operations by composing the two ones. Actually, XOR is the addition modulo 2, while AND is the multiplication modulo 2. Thus if we use use $\binom{4}{2}$-sharing to represent a bit, we can implement XOR and AND similar to addition and multiplication of integer operations in $\mathbb{Z}_{2^n}$.

4.4 Derived POs

We can compose multiple basic POs and form more complex derived POs commonly used in machine learning algorithms. For example, to compute a *ReLu* function $f(x) = \max(0, x)$ which is commonly used as a activation function in neural networks, we can first extract the most significant bit of $-x$ (which indicates x is positive or not), then use the OT protocol to get $f(x)$. For *division*, we can use the Newton-Raphson algorithm [49] to approximate the result. Similarly, to implement the logistic function $f(x) = \frac{1}{1+e^{-x}}$, we can use the Euler method [43]. Another alternative implementation for the logistic function is the piecewise function in [33, 35]. We also implement other common maths functions using similar numerical methods, such as *sqrt*, *log*, *exp* and *max_pooling* in a similar way. With these basic POs and derived POs, we can further implement complex algorithms as usual.

4.5 POs for performance optimization

We provide the following three sets of POs whose functionality is already covered by the basic POs, but the separate versions can significantly improve performance in certain cases. Programmers can use these POs directly.

Batching POs. *Batch up* is a commonly used optimization in MPC frameworks [7, 14, 51], which batches up independent data transfers among the servers and thus reduce the fixed overhead. Array POs natively support batch up. And as many machine learning algorithms heavily utilize array operations, this optimization reduces communication rounds and can improve performance significantly.

Multiply by public variables. In a case where an operation involves both public and private variables, we can optimize performance by revealing the public variables. Multiplication benefits from the optimization the most, as the servers only need to multiply their shares by the public variables directly and there is no necessary communication.

Dot and outer product. Dot and outer product of matrices are frequently used in common machine learning algorithms. For example, logistic regression and neural networks use dot product for forward propagation, represented as $Y = W \cdot X + b$. Outer product is often used for calculating gradients. While implementing them using for-loops, there are too many duplicated transfers for each element, as each element will be multiplied by several other elements in a multi-dimensional situation. We thus provide built-in optimized dot and outer product. Specifically, for two private matrices $[\![A]\!]$ and $[\![B]\!]$, we can calculate the dot product as $[\![A]\!] \cdot [\![B]\!] = A_1 \cdot B'_1 + A_2 \cdot B'_2 + A_a \cdot B'_a + A_b \cdot B'_b$. This optimization significantly reduces communication cost. As an example, given two $n \times n$ matrices, a for-loop for dot product triggers n^3 multiplications and the communication complexity is $O(n^3)$, while the optimized one only incurs communication complexity of $O(n^2)$.

```
x = privpy.ss(clientID)
def logistic(x, start, iter_cnt):
  result = 1.0 / (1 + math.exp(-start))
  deltaX = (x - start) / iter_cnt
  for i in range(iter_cnt):
    derivate = result * (1 - result)
    result += deltaX * derivate
  return result
result = logistic(x, 0, 100) # main()
result.reveal()
```

Figure 3: Example PrivPy code: logistic function.

```
import privpy as pp
x = ... # read data using ss()
factor,gamma,lamb,iter_cnt = initPublicParameters()
n,d = x.shape
P = pp.random.random((n,factor))
Q = pp.random.random((d,factor))
for _  in range(iter_cnt):
  e = x - pp.dot(P,pp.transpose(Q))
  P1 = pp.reshape(pp.repeat(P,d,axis=0),P.shape[:-1] +
   (d,P.shape[-1]))
  e1 = pp.reshape(pp.repeat(e,factor,axis=1),e.shape +
   (factor,))
  Q1 = pp.reshape(pp.tile(Q,(n,1)),(n,d,factor))
  Q += pp.sum(gamma * (e1 * P1 - lamb * Q1),axis = 0)/n
  Q1 = pp.reshape(pp.tile(Q,(n,1)),(n,d,factor))
  P += pp.sum(gamma * (e1 * Q1 - lamb * P1),axis = 1)/d
P.reveal(); Q.reveal()
```

Figure 4: Example PrivPy code: matrix factorization.

5 FRONT-END AND OPTIMIZATIONS

We now introduce the design and implementation of the programming interfaces. Our goal is to provide intuitive interfaces and automatic optimizations to avoid steep learning curves and enable programmers to focus on the machine learning algorithm itself.

5.1 PrivPy Front-end Features

A PrivPy program is a valid Python program with NumPy-style data type definitions. We use three real code segments to illustrate the PrivPy features essential to implementing data mining algorithms.

Fig. 3 shows a PrivPy program that computes the logistic function $f(x) = 1/(1 + e^{-x})$ using the Euler method [43]. Fig. 4 shows an extra example of matrix factorization, which decomposes a large private matrix x to two latent matrices P and Q. Lastly, Fig. 5 shows an example of neural network inference.

Basic semantics. Unlike many domain-specific front-ends [7, 20, 41], which require the programmers to have knowledge about cryptography and use customized languages, the program itself (lines

```
import privpy as pp
x = ... # read data using ss()
W, b = ... # read model using ss()
for i in range(len(W)):
  x = pp.dot(W.T, x) + b
  x = pp.relu(x)
res = pp.argmax(x, axis=1)
res.reveal()
```

Figure 5: Example PrivPy code: neural network inference.

2-9) is a plain Python program, which can run in a raw Python environment with cleartext input, and the user only needs to add two things to make it private-preserving in PrivPy: (*i*) Declaring the private variables. Line 1 declares a private variable `x` as the input from the client *clientID* using the `ss` function. (*ii*) Getting results back. The function `reveal` in line 10 allows clients to recover the cleartext of the private variable. Programmers not familiar with cryptography, such as machine learning programmers, can thus implement algorithms with minimal effort.

All operations support both scalar and array types. PrivPy supports scalars, as well as arrays of any shape. Supporting array operations is essential for writing and optimizing machine learning algorithms which rely heavily on arrays. While invoking the `ss` method, PrivPy detects the type and the shape of `x` automatically. If `x` is an array, the program returns an array of the same shape, containing the function on every element in `x`. Following the NumPy [44] semantics, we also provide *broadcasting* that allows operations between a scalar and an array, as well as between arrays of different shapes, two widely used idioms. That is why the `logistic` function in Fig. 3 works correctly even when `x` is a private array. As far as we know, existing MPC front-ends, such as [7, 14, 41, 51], do not support such elegant program. For example, PICCO [51] only supports operations for arrays of equal shape.

Private array types. Array operations are pretty common in machine learning algorithms. The private array class in PrivPy encapsulates arrays of any shape. Users only need to pass a private array to the constructor, then the constructor automatically detects the shape. Like the array type in Numpy [44], our private array supports *broadcasting*, i.e. PrivPy can handle arithmetic operations with arrays of different shapes by "broadcasting" the smaller arrays (see [22] for details). For example, given a scalar x, a 4×3 array A, a $2 \times 4 \times 3$ array B and a $2 \times 1 \times 3$ array C, the expressions $x \bigodot A$, $A \bigodot B$ and $B \bigodot C$ are all legal in PrivPy, where $\bigodot$ can be $+, \times$and $>$ etc. Note that in PrivPy, the above variables can be either public or private. With broadcasting, programmers can write elegant machine learning algorithms regardless of the shapes of the inputs and model parameters.

We also implement most of the ndarray methods of Numpy, with which application programmers can manipulate arrays conveniently and efficiently, except for the methods related with IO (we leave IO as the future work). Table 1 lists the ndarray methods we have implemented (see [23] for details of `numpy.ndarray`).

Broadcasting and ndarray methods are essential for implementing common machine learning algorithms which usually handle arrays of different shapes.

Both Fig. 4 and Fig. 5 demonstrate *ndarray* methods in PrivPy. Users can implement the algorithms in plain Python, then just replace the *Numpy* package with *PrivPy* package and add private variables declaration. Actually, by replacing all `privpy` with `numpy`, the main parts of Fig. 4 and Fig. 5 can run directly in raw Python environment with cleartext inputs.

Support for large arrays. Mapping the data onto secret shares unavoidably increases the data size. Thus, real-world datasets that fit in memory in cleartext may fail to load in the private version. For example, the $1,000,000 \times 5,048$ matrix require over 150GB memory.

Automatic code rewriting. With the program written by users, the interpreter of our front-end parses it to basic privacy-preserving operations supported by the back-end, and the optimizer automatically rewrites the program to improve efficiency (see Section 5 for details). This optimization can help programmers avoid performance "pit falls" in MPC situation.

5.2 Implementations

Based on Plain Python Interpreter. We write our backend in C++ for performance, and we implement our frontend in Python to keep python compatibility. The backend is linked to the frontend as a library on each server to reduce the overhead between the frontend and backend. During a execution task, the same Python code is interpretered on each server and client in parallel.

NumPy-style data type definitions and operator overloading. We define our own data types `SNum` and `SArr`, to represent the secret numbers and arrays, respectively. Then we overload operators for private data classes, so standard operators such as $+, -, *, >, =$ work on both private and public data. The implementation of these overloaded operators chooses the right POs to use based on data types and the sizes at runtime.

Automatic disk-backed large arrays. We provide a `LargeArray` class that transparently uses disks as the back storage for arrays too large to fit in memory.

5.3 Code analysis and optimization

Comparing to the computation on cleartext, private operations have very distinct cost, and many familiar programming constructs may lead to bad performance, creating "performance pitfalls". Thus, we provide aggressive code analysis and rewriting to help avoid these pitfalls. For example, it is fine to write an element-wise multiplication of two vectors in plain Python program.

```
for i in range(n): z[i] = x[i] * y[i]
```

However, this is a typical anti-pattern causing performance overhead due to the n multiplications involved, comparing to a single array operation (Section 4.5). To solve the problem, we build a source code analyzer and optimizer based on Python's abstract syntax tree (AST) package [36]. Before the servers execute the user code, our analyzer scans the AST and rewrites anti-patterns into more efficient ones. In this paper, we implement three examples:

For-loops vectorization. Vectorization [46] is a well-known complier optimization. This analyzer rewrites the above for-loop into a vector form $\vec{z} = \vec{x} * \vec{y}$. The rewriter also generates code to initialize the vector variables.

fixed-point multiplication	comparison
10, 473, 532	128, 2027

Table 2: Throughput (ops/second) of fundamental operations over $\mathbb{Z}_{2^{128}}$ in the LAN setting.

Common factor extraction. We convert expressions with pattern $x * y_1 + x * y_2 + \cdots + x * y_n$ to $x * (y_1 + y_2 + \cdots + y_n)$. In this way, we reduce the number of $\times$ from n to 1, saving significant communication time.

Common expression vectorization. Programmers often write vector expressions explicitly, like $x_1 * y_1 + x_2 * y_2 + \cdots + x_n * y_n$, especially for short vectors. The optimizer extracts two vectors $\vec{x} = (x_1, x_2, \ldots, x_n)$ and $\vec{y} = (y_1, y_2, \ldots, y_n)$, and rewrite the expression into a vector dot product of $\vec{x} \cdot \vec{y}$. Note that $x_1, x_2, \ldots, x_n$ do not have to be the same shape, as PrivPy supports batch operations with mixed shapes.

Reject for unsupported statements. We allow users to write legal Python code that we cannot run correctly, such as branches with private conditions (actually, most MPC tools do not support private conditions [30, 51], or only support limited scenarios [50, 51]). In order to minimize users' surprises at runtime, we perform AST-level static checking, then reject unsupported statements at the initialization phase and terminate with an error.

6 EVALUATION

Testbed. We run our experiments on four Amazon EC2 virtual machines. All machines are of type `c5.2xlarge` with 8 Intel Xeon Platinum 8000-series CPU cores and 64 GB RAM. Each machine has a 1 GB Ethernet adapter running in full-duplex mode. In our experiments, we consider two network settings: a LAN setting where each virtual machine has 10Gbps incoming and outgoing bandwidth, and a WAN setting where the bandwidth of each virtual machine is 50 Mbps and the RTT latency is 100 ms.

Parameter setting. All arithmetic shares are over $\mathbb{Z}_{2^{128}}$, and we set $d = 40$, which means the scaling factor is 2^{40}. We repeat each experiment 10 times and report the average values.

PrivPy implementation. We implement the front-end of PrivPy with Python, and use C++ to implement our computation engine. And we use the built-in `__int128` type of `gcc` to implement 128-bit integers. We compile the C++ code using `g++ -O3`, and wrap it into Python code using the Boost.Python library [8]. We use SSL with 1024-bit keys to protect all communications.

6.1 Microbenchmarks

We first perform microbenchmarks in the LAN setting to show the performance of basic operations and the benefits of optimizations.

Basic operations. PrivPy engine supports efficient fundamental operations, including addition, fixed-point multiplication and comparison. Addition can be done locally, while multiplication and comparison involve communication. Thus we demonstrate the performance of the latter two, as Table 2 shows.

Client-server interaction. We evaluate the performance of the secret sharing process *ss*, with which the clients split raw data to secret shares and send them to the servers, and the reverse process *reveal*, with which the clients receive the shares from the servers

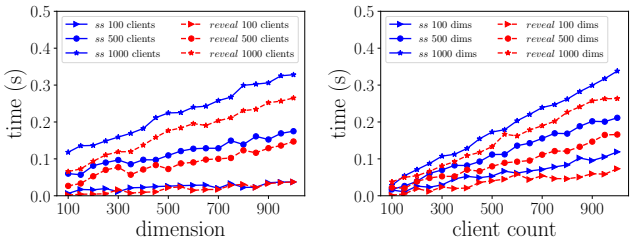

Figure 6: Performance of *ss* and *reveal*.

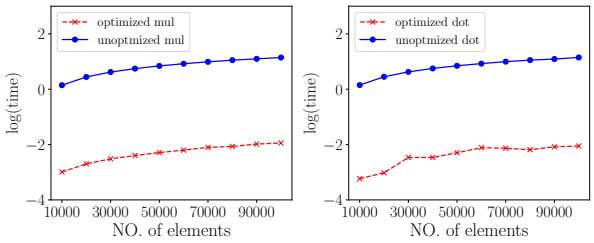

Figure 7: The optimization of doing operations in batch.

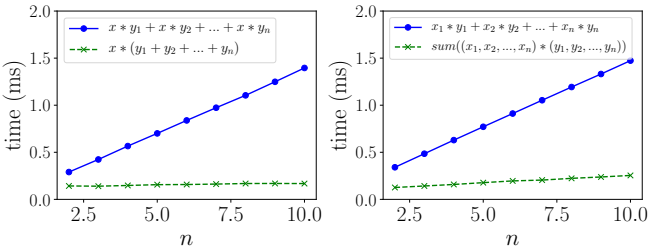

Figure 8: Code optimizer performance.

and recover them to the plaintexts. We evaluate the time (including computation and communication) with different numbers of clients and dimensions, assuming that each client holds an accordingly dimensional vector. Figure 6 shows that even with 1000 clients and 1000-dimension vectors, it takes only less 0.3 seconds for the servers to collect/reveal all the data from/to all the clients.

Effects of batch operations. We evaluate the effectiveness of batching up, using two common operations: *element-wise multiplication* and *dot product* on vectors. For multiplication, we batch up the communication of independent operations, while for dot product of two m-dimensional vectors, we only need to transfer the dot-producted shares and the communication cost is reduced from $O(m)$ to $O(1)$. We vary the number of elements and measure the time consumption, and Fig. 7 shows the result (the y-axis is the logarithm of time). Both cases show benefits over 1000×.

Effects of code optimizations. We evaluate the common factor extraction and expression vectorization. As these hand-written anti-patterns are usually small, we range the expression size from 2 to 10. Figure 8 shows that more than 4× performance improvement for five-term expressions in both situations.

Disk-backed large array performance. The PrivPy front-end provides a class `LargeArray` to automatically handle the large arrays that are too large to fit in the memory. As `LaregeClass` uses disks as back storage, we should consider the effect of the disk IO time to the overall performance. To evaluate the performance

	communication + computation	disk IO	total
single	0.38	0.7	1.08
batched	0.172	0.574	0.746

Table 3: Time (milliseconds) for dot product of a large array.

batch size	LR	MF
single	0.027	0.037
batched	0.026	0.042

Table 4: Time (seconds) for real algorithms with large arrays.

of `LargeArray`, we use the Movielens dataset [19] which contains 1 million movie ratings from thousands of users. We encode the dataset to a $1,000,000 \times 5048$ matrix, which requires 150GB memory space in each machine. We then perform a dot product of a batch of elements in the dataset and a 5048-dimensional vector. We evaluate the performance by varying the batch size and *randomly* choosing a batch of items in the dataset. As Table 3 shows, the disk IO becomes the main cost. The reason is that the program should sequentially scan the large array stored in the disk to retrieve the randomly choosen batch.

We then apply `LargeArray` to the training of real algorithms: logistic regression (LR) which is trained using SGD, and matrix factorization (MF) [5] which decomposes a large matrix into two smaller latent matrices for efficient prediction (in this paper, we decompose each $m \times n$ matrix to a $m \times 5$ matrix and a $5 \times n$ matrix). Table 4 shows the result.

6.2 Performance of real algorithms

The focus of PrivPy is the *algorithm performance* on real big datasets. We present our experience in common algorithms, including *logisgtic regression (LR)*, *matrix factorization (MF)* and *neural network (NN)*, using both ABY3 and our backend. ABY3 engine has several configuration options, we use the most performance-optimized options of ABY3 (semi-honest assumption without precomputation) in all the evaluations.

We perform our evaluation in both the LAN setting and the WAN setting, and use the MNIST dataset [28] which includes 70,000 labeled handwritten digits [12] with 28×28 pixels each. We evaluate the performance for both training and inference. And as our front-end supports both engines, we run *the same* algorithm codes written with Python on the two engines.

Table 5 shows the average time consumed by 1 iteration of training. The logistic regression and matrix factorization is constructed as above, and the neural network (NN) has a 784-dimensonal input layer, two 128- hidden layers and a 10-dimensional output layer.

For inference, we in addition evaulate the LeNet-5 [27] model to demonstrate convonlutional neural network (CNN). LeNet-5 has a 784-dimension input layer, 3 convolutional layers with a 5×5 kernel, 2 sum-pooling layers, 3 sigmoid layers, 1 dot product layer, 1 Radial Basis Function layer, and an *argmin* function on a 10-dimension vector to get the output. Then based on the LeNet-5 model, we add a batch normalization [21] layer to each sigmoid layer to get a CNN+BN model. The evaluation result is as Table 6 shows.

From the evaluation results, we can see that our computation engine performs better than ABY3 for both training and inference, especially in the WAN setting. This is because, although both ABY3 and our computation engine require no precomputation and have the same communication cost for each server, ABY3 requires 1 more round than our computation engine for fixed-point multiplication, thus causes lower performance.

batch size	engine	LAN			WAN		
		LR	MF	NN	LR	MF	NN
single	ABY3	5.2e-3	7.4e-3	1.8e-2	2.16	0.62	1.27
	PrivPy	5.3e-3	7.1e-3	1.7e-2	2.61	0.37	1.16
batched	ABY3	3.94	5.72	58.1	7.53	18.6	637
	PrivPy	3.92	5.67	52.5	7.3	13.2	554

Table 5: Time (seconds) for training of real algorithms with different engines.

batch size	engine	LAN			WAN		
		NN	CNN	CNN +BN	NN	CNN	CNN +BN
single	ABY3	1.3e-2	9.6e-2	0.16	2.43	6.83	8.07
	PrivPy	1.3e-2	9.6e-2	0.17	2.49	7.64	8.07
batched	ABY3	1.45	12.6	13.2	8.12	58.9	59.5
	PrivPy	1.38	12.02	12.2	7.22	56.3	57.9

Table 6: Time (seconds) for inference of real algorithms with different engines.

	LR	MF	NN	CNN	CNN+BN
lines	42	25	9	83	87
time	0.7	0.5	0.1	1.5	1.5

Table 7: Lines of codes and time (hours) for implementing real algorithms.

Finally, we stress the usability of our front-end. Table 7 shows the lines of codes for each algorithm and the time for a student who focuses on data mining but is unfamiliar with cryptography to write each algorithm using our front-end.

7 CONCLUSION AND FUTURE WORK

Over thirty years of MPC literature provides an ocean of protocols and systems great on certain aspects of performance, security or ease of programming. We believe it is time to integrate these techniques into an application-driven and coherent system for machine learning tasks. PrivPy is a framework with top-down design. At the top, it provides familiar Python-compatible interfaces with essential data types like real numbers and arrays, and use code optimizer/checkers to avoid common mistakes. In the middle, using an intermediary for storage and communication, we build a composable PO system that helps decoupling the front-end with backend. At the low level, we design new protocols that improve computation speed. PrivPy shows great potential: it handles large data set (1M-by-5K) and complex algorithms (CNN) fast, with minimal program porting effort.

PrivPy opens up many future directions. Firstly, we are improving the PrivPy computation engine to provide active security while preserving high efficiency. Secondly, we would like to port existing machine learning libraries to our front-end. Thirdly, we will support more computation engines. Fourthly, although we focus on MPC in this work, we will introduce randomization to protect the final results [29, 39]. Last but not least, we will also improve fault tolerance mechanism to the servers.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science fundation of China (NSFC) Grant 61532001, Tsinghua Initiative Research Program Grant 20151080475, and gift funds from Huawei, Ant Financial and Nanjing Turing AI Institute.

Again, we acknowledge the significant contributions of the coauthors in the footnote of the first page.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, and Michael Isard. 2016. TensorFlow: a system for large-scale machine learning. (2016).

[2] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. 2018. Generalizing the SPDZ Compiler For Other Protocols. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 880–895.

[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM Sigsac Conference on Computer and Communications Security*. 805–817.

[4] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *CCS '08*. ACM. https://doi.org/10.1145/1455770.1455804

[5] Arnaud Berlioz, Arik Friedman, Mohamed Ali Kaafar, Roksana Boreli, and Shlomo Berkovsky. 2015. Applying Differential Privacy to Matrix Factorization. In *The ACM Conference*. 107–114.

[6] Dan Bogdanov, Peeter Laud, and Jaak Randmets. 2014. Domain-polymorphic programming of privacy-preserving applications. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*. ACM, 53.

[7] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*. Springer, 192–206.

[8] Boost. 2018. Boost C++ Libraries. http://www.boost.org/.

[9] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine Learning Classification over Encrypted Data.. In *NDSS*.

[10] Octavian Catrina and Sebastiaan De Hoogh. 2010. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks*. Springer, 182–199.

[11] Octavian Catrina and Amitabh Saxena. 2010. Secure computation with fixed-point numbers. In *International Conference on Financial Cryptography and Data Security*. Springer, 35–50.

[12] Zong Chen. 2000. Handwritten Digits Recognition.. In *International Conference on Image Processing, Computer Vision, & Pattern Recognition, Ipcv 2009, July 13-16, 2009, Las Vegas, Nevada, Usa, 2 Volumes*. 690–694.

[13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. 2013. Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits. In *European Symposium on Research in Computer Security*. Springer, 1–18.

[14] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*. Springer, 643–662.

[15] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated synthesis of optimized circuits for secure computation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1504–1517.

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

[17] Duan, Yitao and Canny, John and Zhan, Justin. 2010. P4P: Practical Large-scale Privacy-preserving Distributed Computation Robust Against Malicious Users. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association.

[18] Shai Halevi and Victor Shoup. 2014. Algorithms in helib. In *International Cryptology Conference*. Springer, 554–571.

[19] F Maxwell Harper and Joseph A Konstan. 2016. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* (2016).

[20] Wilko Henecka, Ahmad-Reza Sadeghi, Thomas Schneider, Immo Wehrenberg, et al. 2010. TASTY: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 451–462.

[21] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*. 448–456.

[22] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2011–. numpy.ndarray. https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html.

[23] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2011–. numpy.ndarray. https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html.

[24] Liina Kamm and Jan Willemson. 2015. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security* 14, 6 (2015), 531–548.

[25] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. 2012. Billion-Gate Secure Computation with Malicious Adversaries.. In *USENIX Security Symposium*, Vol. 12. 285–300.

[26] Toomas Krips and Jan Willemson. 2014. Hybrid model of fixed and floating point numbers in secure multiparty computations. In *International Conference on Information Security*. Springer, 179–197.

[27] Yann LeCun et al. 2015. LeNet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet* (2015).

[28] Yann Lecun and Corinna Cortes. 2010. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist.

[29] Yi Li, Yitao Duan, and Wei Xu. 2017. PEM: Practical Differentially Private System for Large-Scale Cross-Institutional Data Mining.. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer.

[30] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 359–376.

[31] Jian Liu, Mika Juuti, Yao Lu, and N Asokan. 2017. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 619–631.

[32] Wenjie Lu, Shohei Kawasaki, and Jun Sakuma. 2016. Using Fully Homomorphic Encryption for Statistical Analysis of Categorical, Ordinal and Numerical Data. *IACR Cryptology ePrint Archive* 2016 (2016), 1163.

[33] Payman Mohassel and Peter Rindal. 2018. ABY 3: a mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 35–52.

[34] Payman Mohassel, Mike Rosulek, and Ye Zhang. 2015. Fast and Secure Three-party Computation:The Garbled Circuit Approach. In *The ACM Sigsac Conference*. 591–602.

[35] P. Mohassel and Y. Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. 19–38. https://doi.org/10.1109/SP.2017.12

[36] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. 2005. Understanding source code evolution using abstract syntax tree matching. In *International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, Usa, May*. 1–5.

[37] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[39] Martin Pettai and Peeter Laud. 2015. Combining differential privacy and secure multiparty computation. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 421–430.

[40] Michael O. Rabin. 1981. *How to exchange secrets by oblivious transfer*. Technical Report TR-81. Aiken Computation Laboratory, Harvard University.

[41] Axel Schropfer, Florian Kerschbaum, and Gunter Muller. 2011. L1-an intermediate language for mixed-protocol secure computation. In *Computer Software and Applications Conference, IEEE 35th Annual*. 298–307.

[42] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979).

[43] Josef Stoer and Roland Bulirsch. 1980. Introduction to numerical analysis. *Math. Comp.* 24, 111 (1980), 749.

[44] Stéfan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

[45] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[46] M Weinhardt and W Luk. 2001. Pipeline vectorization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 20, 2 (2001), 234–248.

[47] Xiaodan Wu, Chao Hsien Chu, Yunfeng Wang, Fengli Liu, and Dianmin Yue. 2007. Privacy Preserving Data Mining Research: Current Status and Key Issues. *Lecture Notes in Computer Science* 4489 (2007), 762–772.

[48] Andrew C. Yao. 1982. Protocols for secure computations. *Foundations of Computer Science Annual Symposium on* (1982), 160–164.

[49] Tjalling J. Ypma. 1995. Historical Development of the Newton-Raphson Method. *Siam Review* 37, 4 (1995), 531–551.

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

[51] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: a general-purpose compiler for private distributed computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 813–826.