# Manticore: Efficient Framework for Scalable Secure Multiparty Computation Protocols

S. Carpov[1], K. Deforth[1], N. Gama[1], M. Georgieva[1], D. Jetchev[1], J. Katz[2], I. Leontiadis[1], M. Mohammadi[1], A. Sae-Tang[1], M. Vuille[1]

[1] Inpher
[2] UMD

**Abstract.** We propose a novel MPC framework, `Manticore`, in the multiparty setting, with full threshold and semi-honest security model, supporting a combination of real number arithmetic (arithmetic shares), Boolean arithmetic (Boolean shares) and garbled circuits (Yao shares). In contrast to prior work [34,32], `Manticore` never overflows, an important feature for machine learning applications. It achieves this without compromising efficiency or security. Compared to other overflow-free recent techniques such as `MP-SPDZ` [17] that convert arithmetic to Boolean shares, we introduce a novel highly efficient modular lifting/truncation method that stays in the arithmetic domain. We revisit some of the basic MPC operations such as real-valued polynomial evaluation, division, logarithms, exponentials and comparisons by employing our modular lift in combination with existing efficient conversions between arithmetic, Boolean and Yao shares. Furthermore, we provide a highly efficient and scalable implementation supporting logistic regression models with real-world training data sizes and high numerical precision through PCA and blockwise variants (for memory and runtime optimizations). On a dataset of 50 million rows and 50 columns distributed among two players, it completes in one day with at least 10 decimal digits of precision. Our logistic regression solution placed first at Track 3 of the annual iDASH'2020 Competition. Finally, we mention a novel oblivious sorting algorithm built using `Manticore`.

## 1 Introduction

**Motivation and background**. Multiparty computation (MPC) is a method for cryptographic computing allowing several parties holding private data to evaluate a public function on their aggregate data while revealing only the output of the function and nothing else. Recent advances in the area make these protocols practical [34,32,6,15,26,27,9,40,36,17,25,3]. Such privacy-preserving computations have been explored in finance [8], bioinformatics [14] and other industry verticals in a variety of use cases including machine learning applications such as linear regressions and logistic regressions [34], SVMs [39], clustering algorithms [33], decision trees [18,12], neural networks [40,24,19,30].

The problem of designing and implementing a highly efficient (in terms of runtime, memory and communication), numerically stable and robust MPC library is thus of central importance for supporting the above functionalities and applications.

**Prior work.** Several MPC protocols and libraries have been proposed in the literature and implemented. `SecureML` [34] provides a practical method in the two-party setting using a combination of arithmetic, Boolean and Yao shares. The method reduces floating point arithmetic with fixed precision to the evaluation of additions, subtractions and rounded divisions on fixed-size integer types (integers modulo $2^{64}$) via a truncation of decimal numbers. `SecureML` was inpired by the prior framework `ABY` [16] (whose security model is 1-out-of-3 corrupted players without a dealer) except that it has been optimized for the vectorized setting. The method is extended to the multiparty setting in [32]. Another recent extension of `ABY` has been constructing efficient protocols for several primitives such as scalar products, matrix multiplications, comparisons, maxpool, and equality testing [35].

Several works [34,32,16,35] are based on Beaver multiplication [5] with local rounding at the price of a small probability of an overflow on the plaintext value. When such an overflow occurs, it is significant. Even if its amplitude has been mitigated since SecureML [34], it still changes the most significant bits of the result in, e.g., ABY3 [32]. Since the probability of overflow is computed on a per-coefficient basis, in a machine learning scenario such as logistic regression, linear regression or neural networks, the presence of at least one destructive overflow gets multiplied by the size of the feature matrix and thus, becomes non-negligible.

The problem of overflows during truncation has recently been addressed in `edaBits` protocol [17] (see also [25]), extending the classical `SPDZ` protocol [15]. The main idea is the use of a new truncation method (a logical right shift operation) in both the semi-honest and the malicious models based on conversions from arithmetic to Boolean shares as well as oblivious comparisons. `SCALE-MAMBA`[3] is another extension of `SPDZ-2` [26,28]. It only supports computations modulo a prime (and not modulo a power of two), garbled circuits and secret-sharing binary computations with malicious security in the honest majority model. The dishonest majority model is only implemented using homomorphic encryption. Yet, the strongest security model (full-threshold, active security with abort and without a dealer) is only possible for very small datasets. The protocols are most practical in the 1-out-of-3 setting without a dealer (for three players) or for $n$ players in the full-threshold setting for secret shares only (with a dealer or with FHE) and passive security. Most of these protocols propose linear and logistic regression but mainly for full-rank and feature scaled input datasets.

Finally, `Sharemind` [6,7] together with subsequent extensions such as [38] for both arithmetic, Boolean secret shares as well as garbled circuits is a framework providing 1-out-of-3 and full threshold. Various algorithms for oblivious sorting

[3] https://github.com/KULeuven-COSIC/SCALE-MAMBA

have been proposed in the literature built on top of `Sharemind` [23] (implementing Batcher's merge sort), [21,13].

**Our contributions.** In this work, we present a novel framework, `Manticore`, for real number and Boolean arithmetic with high numerical precision in the MPC setting. We only focus on the semi-honest security model with an offline dealer, with full-threshold security across an arbitrary number of players. We split the computation into offline and online phases. The offline phase (independent of the input data) is performed by a trusted dealer and is compatible with stronger models and the verifiability of this phase can be achieved via standard techniques such as oblivious transfer and cut-and-choose. The offline phase generates precomputed data (e.g triples or masks) and distributes the secret shares of these masks to the parties. The online phase computes additive shares of the result.

We are primarily interested in efficient implementation, numerical stability and robustness against data unbalanced-ness and singularities. Our implementation uses well-known secret sharing protocols starting from Beaver multiplication and efficient conversions between arithmetic and Boolean shares. We represent real numbers as fixed-point numbers using 64-bit and 128-bit integers on the backend (Sections 3.1 and 3.2). A major improvement is coming from our algorithms for modular lifts and truncations in Section 3.3 using masking data precomputed in the offline phase. This eliminates the non-zero probability of overflow compared to [34,32] and provides more efficient analogues of the logical right shift operator of [17] avoiding the use of two oblivious comparisons (at least 8 times faster and 12 times less triples (see Table 1)).

Besides the almost classical tensor-like approach used in multiple prior works where one round of communication is performed per tensor as opposed to per individual coefficient, we improve communication complexity by using seeded precomputed data (i.e., sending a seed instead of a full mask to reconstruct the mask in the online phase); see Section 6 for the details.

Our logistic regression algorithm is presented in details in Section 5.1. For better numerical stability as well as robustness against singular features matrices, we use principal component analysis as well as internal normalization. Faster convergence is achieved via second-order optimization methods (IRLS, or Newton–Raphson) - a useful feature in the MPC setting. High precision is obtained via a uniform approximation of the sigmoid function via Fourier series based on the ideas of [9] (see also Section 4.1), thus, achieving the prescribed high numerical precision. Major runtime and memory improvements for large-scale feature matrices is done via blockwise versions of the algorithm (a row-blockwise variant improving the memory overhead and a column-wise variant used for improving the running time). Our experiments for training a logistic regression model with 1–50M samples and up to 100 balanced or unbalanced features with possible singularities, yield more than 7 decimal digits of precision versus the same plaintext computation, which significantly outperforms state-of-the-art implementations. `Manticore` based MPC logistic regression solution was

recognized for delivering the best Federated Learning-based Cancer Prediction Model at Track 3 of the annual iDASH'2020[4] with the highest model accuracy with the lowest runtime.

Besides computing with real numbers, `Manticore` supports Boolean operations and garbled circuits as well as conversion algorithms between arithmetic and Boolean/garbling shares (described in Section 3.4). Our hybrid approach for private comparisons that leverages both Sklansky adders and ripple-carry adders is described in Section 4.2). We also apply these Boolean primitives in the context of private division and other operations (see Section 4.3). Data alignment techniques are employed for faster memory access (also available for the real numbers support).

Finally, in Section 5.2 we discuss a novel approach to oblivious sorting based on the `Manticore` framework. Compared to prior art, it has better memory and communication complexities thanks to the implementation of secure shuffling (via Benes networks) and the multi-pivot approach suitable for better parallelism of the underlying oblivious comparisons.

## 2 Manticore MPC protocol

### 2.1 Protocol participants and phases

An MPC algorithm is a computational procedure that is split both temporally and physically across multiple participants. In our setting, a set of private data owners provide the input data and a data analyst (DA) is interested in evaluating a public function $f$ over the input data via a multiparty computation protocol. In this protocol, a set of computing parties (CP) receive shares of the input values (the CP's are deployed inside the perimeter of the corresponding private data owners), a trusted dealer (TD) helps the CP's compute $f$ by generating and secret-sharing among the parties random precomputed data (e.g., Beaver triples). We assume that the dealer does not participate in the online phase (the phase when the CP's operate on the input data) and thus, it sees neither the local private data, nor the communicated masked data. Furthermore, it does not collude with any of the parties. In the paper, we interchangeably use CPs, parties and players. Our CPs are semi-honest and the security model is full-threshold across an arbitrary number of parties.

In the simplest model, there are $n$ CP's and one TD. The TD executes the *offline phase* of the MPC algorithm that produces the *precomputed data* (also referred to as *triples* or *masks*) by generating and distributing secret shares of these masks to the players (see Figure 1). Each player knows its share of the mask, but none of them knows the actual mask value. The offline phase does not depend on the input data. The $n$ CPs then run the *online phase* of the MPC algorithm. It is a joint computation with synchronization steps where CP's can exchange or broadcast messages according to the MPC protocol. The online phase runs within a firewall-ed environment among the players where the trusted dealer has no access (see Figure 1).

[4] http://www.humangenomeprivacy.org/2020/

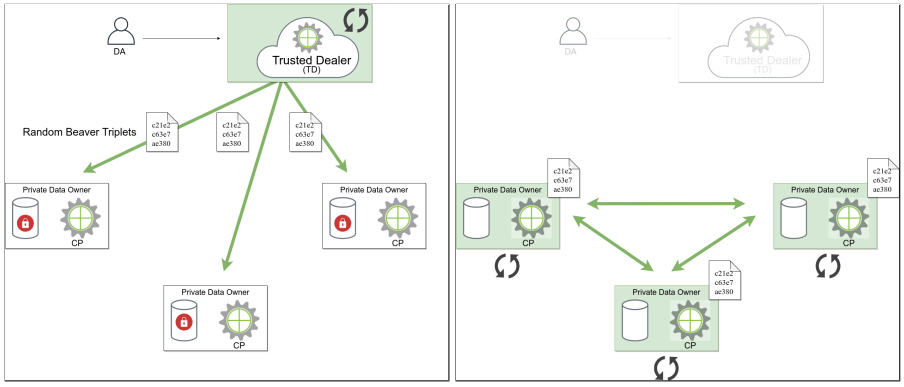



**Fig. 1.** The offline phase in left and the online phase in right

### 2.2 MPC variables and builtins

Our MPC protocol is based on secret sharing:

**Definition 1 (secret sharing).** *If $(G, +)$ is an abelian group, then an element $x \in G$ is said to be secret shared among the $n$ players $P_1, \ldots, P_n$, if every player $P_i$ holds an $x_i$, such that $x_1 + x_2 + \ldots + x_n = x$. We use $[\![x]\!]$ to denote a $n$-tuple of secret shares $(x_1, \ldots, x_n)$.*

The MPC algorithm is the distributed equivalent of a plaintext pseudocode which we can describe as an SSA (static single assignment) graph of MPC-friendly elementary operations. The nodes of the SSA graph correspond to all the immutable variables that occur during the execution, and each player gets a local view (or share) of these variables. We refer to these local views as *MPC variables*. The MPC-friendly elementary operations are *builtins* that take MPC variables (together with some static parameters) as inputs and that produce MPC variables as outputs. Globally, an MPC variable holds all the information about one variable in the SSA, namely: one plaintext value $x$ that can be either public (known by all players, but not to the trusted dealer) or secret-shared $[\![x]\!]$ (each player only knows its share) or more rarely, dealer-generated; one mask $[\![\lambda]\!]$ (known by the dealer, and secret shared among all players); and the optional masked value $a = x + \lambda$ (known by all players, but not by the dealer).

Locally, each CP has a partial view of the MPC variable, which is a structure with 4 optional fields; 1. the public value $x$ (if the variable is publicly revealed); 2. one share $x_j$ of the public value; 3. one share $\lambda_j$ of the variable's mask; 4. the masked value $a$ (if the variable is masked and revealed). The special *mask-and-reveal* operation instructs each player to broadcast the masked share $x_j + \lambda_j$, thus allowing the parties to jointly reconstruct and store the common masked value $a = x + \lambda$. The trusted dealer has access only the plain variable's mask $\lambda$ and all dealer-generated values.

In order to reduce the communication complexity overall by a factor $n$, we use a standard technique based on deterministic pseudorandom number generators (see Supplementary Material 6).

## 3 Representation of Real Numbers and Booleans

A real number $x \in \mathbb{R}$ can be represented as a floating point number $x = 2^e \cdot m \in \mathbb{R}$ where the mantissa $m$ is normalized so that $1/2 \leq |m| < 1$ and the exponent $e \in \mathbb{Z}$. This representation is clearly data-dependent since both $m$ and $e$ depend on the plaintext value $x$, the exponent being $e = \lceil \log_2 |x| \rceil$, thus, making the classical floating-point representation unsuitable for multi-party computation.

On the other hand, a fixed-point representation assumes that the exponent $e$ is fixed independently of the data, but with a risk of overflow or underflow if $e$ is too small or too large. We thus adopt fixed-point approach and we bookkeep a sufficiently good bound on the exponent for each value in the program. This bound is estimated statically, by certain statistical analysis, via a special-purpose compiler. This method keeps track of public bounds, as precisely as possible, on the secret value, without revealing the secret value itself. From the compiler's perspective, the exponent should be thought of as being public whereas the mantissa is private. Yet, as the compiler certainly does not know the secret number $x$, it is impossible for the compiler to determine the exact exponent $e$ that guarantees the above normalization for $m$, but only an upper bound. In practice, the limiting factor is the size of the numerical window $\rho$ (that is, the number of bits of the mantissa $m$; equivalently, the difference between the exponent bound kept by the compiler and the number of binary digits in the fractional part to be kept). The smaller the $\rho$ is, the more efficient the arithmetic is on the backend. In practice, we often use either 64-bit or 128-bit integer arithmetic.

In a related work [2], a floating point representation is used for secure operations on a real data. The mantissa, the exponent, as well an additional bit for the sign and the zero are all secret shared. The floating point approach avoid the overflow issues (especially in the case of multiplication), but require a binary decomposition for all computations, that makes it less efficient comparing to fixed-point representation, where the exponent is public and just the mantissa is secret shared.

### 3.1 Plaintext representation

Motivated by the fixed-point representation used in [34], [32], we define the classes of plaintext values in `Manticore` as follows:

**Definition 2 (plaintext classes).** *Given integers $p_{msb} > p_{lsb}$, the class of plaintext values represented using these parameters is denoted by $\mathcal{P}_{p_{msb},p_{lsb}}$ and is defined to be:*

$$\mathcal{P}_{p_{msb},p_{lsb}} = \{x \in 2^{p_{lsb}} \cdot \mathbb{Z}, |x| \leq 2^{p_{msb}}\} \subset \mathbb{Q}.$$

Here, $p_{\text{msb}}$ and $p_{\text{lsb}}$ indicate the positions of the most significant and the least signficant bits of the plaintext, respectively.

Note that the class $\mathcal{P}_{p_{\text{msb}},p_{\text{lsb}}}$ represents all real (in this case, rational) numbers between $-2^{p_{\text{msb}}}$ and $2^{p_{\text{msb}}}$ with step $2^{p_{\text{lsb}}}$. In some sense, we are using a floating-point backend to represent a class of fixed-point numbers. For $p_{\text{lsb}} = 0$, the class contains integers only. Negative $p_{\text{lsb}}$ means that we are considering bits for the fractional part. The size $\rho = p_{\text{msb}} - p_{\text{lsb}}$ of the numerical window is the number of bits needed to represent the number. The smaller $p_{\text{lsb}}$ is, the higher precision one has for the number and the more bits one needs to represent it. For comparison, [32] and [34] impose a fixed value of $p_{\text{lsb}}$ when performing the truncation while our choice above provides us a bit more flexibility. The idea is that, at least in theory, if one is interested in a given numerical precision for the output, one can assign specific $p_{\text{lsb}}$'s to all intermediate variables in a computation to achieve this output. Thus, any secret variable $z$ in our MPC program is assigned at compile time to a publicly known plaintext class $\mathcal{P}_{p^{(z)}_{\text{msb}},p^{(z)}_{\text{lsb}}}$. For each real-valued operation, e.g., $z = f(x, y)$, we assume that the inputs belong to their corresponding classes and that the output $z$ is rounded to its class: this induces a least significant floating-point error of magnitude $\approx 2^{p^{(z)}_{\text{lsb}}}$ at each step. If the plaintext class of $z$ were wrongly estimated, $|f(x, y)| > 2^{p^{(z)}_{\text{msb}}}$ will result in an *overflow* and a non-zero $|f(x, y)| \ll 2^{p^{(z)}_{\text{lsb}}}$ will be an *underflow*, both yielding an undefined behaviour. Plaintext class misprediction and floating point rounding error propagation can be avoided or controlled by carefully designing end-to-end *numerically stable* or MPC-friendly algorithms, as explained in Section 5.1. We thus assume throughout the paper that all plaintext class are correctly predicted.

### 3.2 Secret shares representation: `Modreal`

To enable MPC computations, we specify how we secret share real numbers represented in the above form:

**Definition 3 (modreal classes).** *Given parameters $M_{msb}$ and $p_{lsb}$, the class used for modular representation of secret shares of real numbers is defined as the finite abelian group*

$$\mathcal{M}_{M_{msb},p_{lsb}} = \left\{ \sum_{i=p_{lsb}}^{M_{msb}-1} m_i 2^i \mod 2^{M_{msb}}, \quad \textit{for} \quad m_i \in \{0,1\} \right\} = 2^{p_{lsb}}\mathbb{Z}/2^{M_{msb}}\mathbb{Z}.$$

The important properly of the class $\mathcal{M}_{M_{\text{msb}},p_{\text{lsb}}}$ (unlike the set of all real numbers $\mathbb{R}$) that will ultimately allow us to achieve information-theoretic security properties of the MPC computations is that it admits a uniform distribution.

Before we define the secret shares, we discuss several natural lifts of $\mathbb{R}/M\mathbb{Z}$ (for a positive integer $M$) to the real numbers that will be used throughout.

- $\texttt{posmod}_M(x)$: is defined as the unique real number $\widetilde{x} \in [0, M)$ such that $\widetilde{x} - x \in M\mathbb{Z}$ (positive representative modulo $M$).

– $\texttt{centermod}_M(x)$: is defined to be the unique real number $\widetilde{x} \in \left[-\frac{1}{2}M, \frac{1}{2}M\right)$ such that $\widetilde{x} - x \in M\mathbb{Z}$. (signed representative modulo $M$)
– $\texttt{quartermod}_M(x)$: is defined as the unique real number $\widetilde{x} \in \left[-\frac{1}{4}M, \frac{3}{4}M\right)$ such that $\widetilde{x} - x \in M\mathbb{Z}$.

When all moduli are powers of 2, `centermod` and `posmod` formalize exactly the difference between signed and unsigned integers in C, where the same sequence of 64 bits on the machine is interpreted either as a signed integer in $[-2^{63}, 2^{63}-1]$ or an unsigned integer in $[0, 2^{64}-1]$, or for that matter, an integer in $[-2^{62}, 3.2^{62}-1]$ in the `quartermod` case. Similarly, a residue $x \in \mathcal{M}_{M_{\text{msb}},p_{\text{lsb}}}$ can be represented by the machine 64-bit integer $2^{64-M_{\text{msb}}}x \pmod{2^{64}}$ or 128-bit integer $2^{128-M_{\text{msb}}}x \pmod{2^{128}}$, hence benefiting from the natural modular overflow of 64-bit (resp. 128-bit) registers for addition and multiplication without having to call any explicit modular reduction.

**Definition 4 (`Modreal` secret shares).** *We say that a number $x \in \mathcal{P}_{p_{msb},p_{lsb}}$ is secret shared in the class $\mathcal{M}_{M_{msb},p_{lsb}}$ as $[\![x]\!]_{M_{msb},p_{lsb}} = (x_1, \ldots, x_n) \in \mathcal{M}^n_{M_{msb},p_{lsb}}$ if*

$$x = \texttt{centermod}_{2^{M_{msb}}}\left(\sum_{j=1}^{n} x_j\right).$$

*The sharing is well-defined if $M_{msb} \geq p_{msb} + 2$ (because $M_{msb} = p_{msb} + 2$ is the smallest exponent distinguishing $2^{p_{msb}}$ from $-2^{p_{msb}}$). Furthermore, if the first $n-1$ shares are uniformly distributed, this sharing is full-threshold unconditionally secure (as $n-1$ shares do not reveal any information about $x$).*

One advantage of the `Modreal` classes is that they are already compatible with real addition and multiplication: if $x \in \mathcal{M}_{M^{(x)}_{\text{msb}},p^{(x)}_{\text{lsb}}}$, $y \in \mathcal{M}_{M^{(y)}_{\text{msb}},p^{(y)}_{\text{lsb}}}$, then the sum $s = x + y$ is defined in any class $\mathcal{M}_{M^{(s)}_{\text{msb}},p^{(s)}_{\text{lsb}}}$ for which $M^{(s)}_{\text{msb}} \leq \min(M^{(x)}_{\text{msb}}, M^{(y)}_{\text{msb}})$, $p^{(s)}_{\text{lsb}} \leq \min(p^{(x)}_{\text{lsb}}, p^{(y)}_{\text{lsb}})$. Similarly, the product $p = xy$ is defined in any class $\mathcal{M}_{M^{(p)}_{\text{msb}},p^{(p)}_{\text{lsb}}}$ for which $p^{(p)}_{\text{lsb}} \leq p^{(x)}_{\text{lsb}} + p^{(y)}_{\text{lsb}}, M^{(p)}_{\text{msb}} \leq \min(M^{(x)}_{\text{msb}} + p^{(y)}_{\text{lsb}}, p^{(x)}_{\text{lsb}} + M^{(y)}_{\text{msb}})$. Unfortunately, these bounds are often too small for secret shares to represent correctly the plaintext underneath: $p_{\text{lsb}}$ can be artificially increased after the sum or product by rounding each share locally: this corresponds the floating point rounding. However, increasing $M_{\text{msb}}$ (above $p_{\text{msb}} + 2$ so that the resulting shares are correct, as per Definition 4) must be done before sum or product on both inputs. This operation, which we call *lift*, is non trivial and requires communication between all the secret-share holders. In Section 3.3, we propose a new lift algorithm that is error-less and efficient.

### 3.3 Lifts with precomputed data and cast operations

Given a plaintext value $x \in \mathcal{P}_{p_{\text{msb}},p_{\text{lsb}}}$ for some parameters $p_{\text{msb}} \geq p_{\text{lsb}}$, secret shares $[\![x]\!]_{M_{\text{msb}},p_{\text{lsb}}}$ in an input modular shares class $\mathcal{M}_{M_{\text{msb}},p_{\text{lsb}}}$ such that

$M_{\text{msb}} \geq p_{\text{msb}} + 2$, as well as an output modular shares class $\mathcal{M}_{M'_{\text{msb}},p'_{\text{lsb}}}$ with $M'_{\text{msb}} > M_{\text{msb}}$, the lifting algorithm needs to compute secret shares $[\![x]\!]_{M'_{\text{msb}},p'_{\text{lsb}}}$ of the same plaintext value $x$. This problem has been subject to many lines of research: most of the techniques involve a conversion to Booleans [17], other techniques stay in the arithmetic domain but have a non-negligible probability of overflow per coefficient [34]. The latter can only be reduced by increasing the bit-length of the secret shares [32]. The lift we propose in this section avoids overflows and has no additional overhead on the secret-share bit length, thus, improving on the prior work [34], [32] where the algorithms are probabilistic and have non-zero probability of failure.

For simplicity, the algorithms are presented on individual scalars, but they extend coefficient-wise to any tensor.

Let $M = 2^{M_{\text{msb}}}$. We start by using the trusted dealer to generate a mask $\lambda \in \mathcal{M}_{M_{\text{msb}},p_{\text{lsb}}}$. Besides computing secret shares $[\![\lambda]\!]_{M_{\text{msb}},p_{\text{lsb}}}$ for $\lambda$, the dealer will extract and secret share one bit of information representing the half-interval containing $\texttt{quartermod}_M(\lambda)$. More precisely, define

$$b_\lambda = \begin{cases} 0 & \text{if } -M/4 \leq \texttt{quartermod}_M(\lambda) < M/4, \\ 1 & \text{if } \quad M/4 \leq \texttt{quartermod}_M(\lambda) < 3M/4. \end{cases}$$

Letting $a = x + \lambda \in 2^{p_{\text{lsb}}}\mathbb{Z}/2^{M_{\text{msb}}}\mathbb{Z}$ be the masked value, one can consider the two lifts (both in $\mathbb{R}$)

$$\widetilde{a}_0 = \texttt{centermod}_M(a) \qquad \text{and} \qquad \widetilde{a}_1 = \texttt{posmod}_M(a).$$

Since $M_{\text{msb}} \geq p_{\text{msb}} + 2$, $\texttt{centermod}_M(x) \in [-M/4, M/4)$, and we have

$$\texttt{centermod}_M(x) + \texttt{quartermod}_M(\lambda) = \begin{cases} \widetilde{a}_0 & \text{if } b_\lambda = 0, \\ \widetilde{a}_1 & \text{if } b_\lambda = 1, \end{cases}$$

which we conveniently rewrite as

$$\texttt{centermod}_M(x) + \texttt{quartermod}_M(\lambda) = \widetilde{a}_0 + b_\lambda(\widetilde{a}_1 - \widetilde{a}_0) \in \mathbb{R}. \tag{1}$$

To get shares of $x$ in the output modular class $\mathcal{M}_{M'_{\text{msb}},p'_{\text{lsb}}}$, one would like to round the above real number to the nearest integer multiple of $2^{p'_{\text{lsb}}}$ and reduce this multiple modulo $2^{M'_{\text{msb}}}$.

To perform this computation, the players first compute the right-hand side of (1), one simply needs precomputed (by the dealer) secret shares of $b_\lambda$ *a priori* in the modular class $\mathcal{M}_{M'_{\text{msb}},p'_{\text{lsb}}}$. For the unmasking, one define

$$\nu := \texttt{roundTo}(\texttt{quartermod}_M(\lambda), 2^{p'_{\text{lsb}}}) \mod 2^{M'_{\text{msb}}},$$

where the function $\texttt{roundTo}(z, 2^\ell)$ takes a real number $z$ and an integer $\ell$ (not necessarily positive) and rounds $z$ to the nearest integer multiple of $2^\ell$. The dealer can secret share $\nu$ in the output modular class among the players.

In fact, one can do slightly better to optimize communication: letting $a_j = \texttt{roundTo}(\widetilde{a}_j, 2^{p'_{\text{lsb}}}) \bmod 2^{M'_{\text{msb}}}$ for $j = 0, 1$, note that the only non-zero bits in the binary representation of $a_1 - a_0$ are the top $M'_{\text{msb}} - M_{\text{msb}}$ bits. It thus suffices for the trusted dealer to secret share the bit $b_\lambda$ in the smaller modular class $\mathcal{M}_{M'_{\text{msb}}-M_{\text{msb}},0}$ instead of $\mathcal{M}_{M'_{\text{msb}},p'_{\text{lsb}}}$ in order for the parties to perform the computation.

In the online phase of the computation, one uses the lifts $a_0$ and $a_1$ to $2^{p'_{\text{lsb}}}\mathbb{Z}/2^{M'_{\text{msb}}}\mathbb{Z}$ and only extracts the top $M'_{\text{msb}} - M_{\text{msb}}$ bits for the oblivious selection in the computation of the above $\texttt{centermod}_M(x) + \texttt{quartermod}_M(\lambda)$.

Let $\texttt{msb}_{M'_{\text{msb}}-M_{\text{msb}}} : \mathbb{Z}/2^{M'_{\text{msb}}}\mathbb{Z} \to \mathbb{Z}/2^{M'_{\text{msb}}-M_{\text{msb}}}\mathbb{Z}$ be the projection keeping only the $M'_{\text{msb}} - M_{\text{msb}}$ most significant bits. One can define the operation

$$\boxtimes\colon 2^{p'_{\text{lsb}}}\mathbb{Z}/2^{M'_{\text{msb}}}\mathbb{Z} \times \mathbb{Z}/2^{M'_{\text{msb}}-M_{\text{msb}}}\mathbb{Z} \to 2^{p'_{\text{lsb}}}\mathbb{Z}/2^{M'_{\text{msb}}}\mathbb{Z}$$

as $x \boxtimes y := \texttt{msb}_{M'_{\text{msb}}-M_{\text{msb}}}(x) \cdot y$, extended with $M_{\text{msb}} - p'_{\text{lsb}}$ zeros at the end.

Below, we describe the offline (Algorithm 1 and online (Algorithm 2) phases of the lifting algorithm:

**Algorithm 1** Lift: Offline phase

**Input:** The mask $\lambda \in 2^{p_{\text{lsb}}}\mathbb{Z}/2^{M_{\text{msb}}}\mathbb{Z}$ of a modular real number
**Output:** Precomputed shares $[\![\lambda]\!]_{M_{\text{msb}},p_{\text{lsb}}}$, $[\![b_\lambda]\!]_{M'_{\text{msb}}-M_{\text{msb}},0}$ and $[\![\nu]\!]_{M'_{\text{msb}},p'_{\text{lsb}}}$.

1: $[\![\lambda]\!]_{M_{\text{msb}},p_{\text{lsb}}} := \texttt{secretShares}(\lambda, \mathcal{M}_{M_{\text{msb}},p_{\text{lsb}}})$
2: Set $b_\lambda = 0$ if $\lambda \in [-2^{M_{\text{msb}}-2}, 2^{M_{\text{msb}}-2})$, $b_\lambda = 1$ if $\lambda \in [2^{M_{\text{msb}}-2}, 3 \cdot 2^{M_{\text{msb}}-2})$.
3: $[\![b]\!]_{M'_{\text{msb}}-M_{\text{msb}},0} := \texttt{secretShares}(b, \mathcal{M}_{M'_{\text{msb}}-M_{\text{msb}},0})$
4: Compute $\nu = \texttt{roundTo}(\texttt{quartermod}_{2^{M_{\text{msb}}}}(\lambda), 2^{p'_{\text{lsb}}}) \mod 2^{M'_{\text{msb}}}$ .
5: $[\![\nu]\!]_{M'_{\text{msb}},p'_{\text{lsb}}} := \texttt{secretShares}(\nu, \mathcal{M}_{M'_{\text{msb}},p'_{\text{lsb}}})$
6: **return** $[\![\lambda]\!]_{M_{\text{msb}},p_{\text{lsb}}}$, $[\![b_\lambda]\!]_{M'_{\text{msb}}-M_{\text{msb}},0}$ and $[\![\nu]\!]_{M'_{\text{msb}},p'_{\text{lsb}}}$.

**Algorithm 2** Lift: Online phase

**Input:** For given parameters $p_{\text{lsb}}, M_{\text{msb}}, p'_{\text{lsb}}, M'_{\text{msb}}$ :
- A pair $([\![x]\!]_{M_{\text{msb}},p_{\text{lsb}}}, [\![\lambda]\!]_{M_{\text{msb}},p_{\text{lsb}}}, a = x + \lambda)$ of a secret shared modular real number and a mask for a plaintext $x \in \mathcal{P}_{p_{\text{msb}},p_{\text{lsb}}}$ and some $p_{\text{msb}} \leq M_{\text{msb}} - 2$,
- Triple shares $[\![b_\lambda]\!]_{M'_{\text{msb}}-M_{\text{msb}},0}, [\![\nu]\!]_{M'_{\text{msb}},p'_{\text{lsb}}}$.

**Output:** Output secret shares $[\![x]\!]_{M'_{\text{msb}},p'_{\text{lsb}}}$ of the same plaintext $x$.

1: Mask and reveal $a = (x + \lambda) \mod 2^{M_{\text{msb}}}$.
2: Player 1 computes $a_0 = \texttt{roundTo}(\texttt{centermod}_{2^{M_{\text{msb}}}}(a), 2^{p'_{\text{lsb}}}) \mod 2^{M'_{\text{msb}}}$
3: All compute $[\![x]\!]_{M'_{\text{msb}},p'_{\text{lsb}}} = a_0 + (a_1 - a_0) \boxtimes [\![b_\lambda]\!]_{M'_{\text{msb}}-M_{\text{msb}},0} - [\![\nu]\!]_{M'_{\text{msb}},p'_{\text{lsb}}}$, where

$$a_1 = \texttt{roundTo}(\texttt{posmod}_{2^{M_{\text{msb}}}}(a), 2^{p'_{\text{lsb}}}) \mod 2^{M'_{\text{msb}}}$$

4: **return** $[\![x]\!]_{M'_{\text{msb}},p'_{\text{lsb}}}$

The method keeps a single round of communication during the online phase and adds only a few binary operations that preserve the overall running time. The new algorithm is always correct, the security is unconditional and works for any number of players.

It is worth comparing our `Modreal` representation and lifting algorithms to the algorithm in the recent work of [17].

First, [17] proposes an MPC integer arithmetic with support for addition, multiplication, left and right shifts. In theory, this is sufficient to represent real numbers by using right shifts as right shifts. Yet, to make this practical, there are several problems to be carefully analyzed and addressed by the user:

- Consistently representing real numbers to integer representation across the computation; in other words, the user has to translate the exponents manually in terms of left or right shifts between the elementary operations
- Addressing potential overflows that might compromise the correctness of the computation.

Our backend enables the programmer to write the algorithm as if it operates on floating point numbers (i.e., without explicitly calling the suitable conversions and shift operators). In fact, the user just needs to specify the ranges and exponents of the input variables and then we automate the estimation and propagation of these exponents to all other variables by our compiler. In addition, we automate and hide from the user the placements of casts (lifts and rounding) throughout the whole program.

Of the above-mentioned shifts, only the logical right shift [17, Fig.9] requires precomputed data. Unlike our approach, this algorithm uses two oblivious comparisons - in steps 1(b) and 2(c), as well as `Boolean`-to-`Arithmetic` conversions in steps 1(c) and 2(d) of loc.cit. while the algorithm proposed above does not go through these expensive conversion methods and the two oblivious comparisons steps. There are two more advantages of the lift/round operations and the underlying representation: 1) the cases are more flexible (i.e., the numerical window in `Manticore` is not fixed, so we can only increase or decrease $M_{\text{msb}}$ or $p_{\text{lsb}}$; 2) `Manticore` allows for automating the estimate of these parameters at compiler time; 3) `Manticore` enables switching from 64 to 128-bit integers. The estimates, given in Table 1, for [17] logical right-shift have been obtained from the oblivious comparisons timings in the honest majority/semi-honest setting [17, Table 7] considering that the logical right shift protocol uses two comparisons.

| Comparison operation | Throughput ops/sec | Triples triple bits per coeff |
|---|---|---|
| `Manticore` lift/round | $6.3M$ | 64 |
| [17] logical right shift | $\leq 0.8M$ | 800 |

**Table 1.** Comparison of Manticore's Lift to [17] Logical right-shift.

Finally, it may seem surprising at a first glance that our approach of representing shares in the modular shares class and modular lifts require in some rare cases interaction even for linear operations like additions/linear combinations (prior work typically implements such operations non-interactively). Only cases for which $M'_{\text{msb}} > M_{\text{msb}}$ need interaction. In prior art, linear operations avoid lifts by keeping $M_{\text{msb}}$ constant and sufficiently large - this trades off the interaction for less precise bounds on $M_{\text{msb}}$ and hence, requires larger numerical windows. In our case, by allowing interactions, we manage to book-keep the optimal upper bounds on $M_{\text{msb}}$ and thus, calculate with shorter numerical windows without compromising numerical precision (i.e., $p_{\text{lsb}}$). This allows us to, e.g., use only 64-bit integers for certain non-linear operations evaluated via Fourier series while keeping the precision high (see Section 4.1). It is also useful when chaining multiple operations where careful optimizations are very important. Our special purpose compiler provides an automated and precise static statistical analysis on the $M_{\text{msb}}$ parameters for all intermediate variables.

### 3.4 Booleans: secret shares and garbled circuits representations

Similarly to the `ABY-3` construction [32], `Manticore` supports conversion between arithmetic and Boolean shares to enable operations such as oblivious comparison that need access to the individual bits. Both Boolean sharing and garbling representations are supported.

Conversion from arithmetic shares $[\![\cdot]\!]_A$ to Boolean shares $[\![\cdot]\!]_B$ (resp. Yao shares $[\![\cdot]\!]_Y$ as introduced in [16, III.(C)] for two parties) is done via the same technique as in [16]: by revealing $a = x + \lambda$ for a uniformly random, dealer-generated mask $\lambda$, plaintext bit-composing $a$, and canceling $\lambda$ using either the dealer-generated Boolean shares $[\![\lambda]\!]_B$ (see also `edaBits` from [17]), or the dealer-generated circuit $a \mapsto a - \lambda$. For conversion from Boolean (resp., Yao) to arithmetic shares, we perform the operations in reverse order.

If $x$ is a Boolean tensor, we use $[\![x]\!]_\oplus$ to denote a $n$-tuple of tensors $(x_1, \ldots, x_n)$ such that $x_1 \oplus \cdots \oplus x_n = x$. If $x_1, \ldots, x_{n-1}$ are independent uniformly random tensors, we refer to $[\![x]\!]_\oplus$ as *secure Boolean secret shares* of $x$. Boolean tensors are mostly used to evaluate component-wise Boolean operations over full columns: we pack each columns so that one clock cycle treats between 8 and 256 bitwise operations. Boolean sharing is the equivalent of additive sharing with coefficient over the field $\mathbb{F}_2$.

For garbled circuits, `Manticore` relies on a free-XOR point-and-permute garbling scheme, emphasizing on the analogy between garbling and secret-sharing: a plaintext bit $\{0, 1\}$ is mapped to $\{0, R\} \in \mathbb{F}_2^{128}$, where $R$ is a global 128-bit *odd* secret[5] known by the dealer - it exactly corresponds to the random global key generated by the garbler and used for computing the labels in the classical free-XOR technique [29]. Each plaintext bit $x$ (resp. $y$) is masked by a 128-bit random mask $\lambda$ (resp. $\mu$) known by the dealer/garbler. The players/evaluators

[5] Meaning that the least-significant bit of $R$ (or the last component of $R$ viewed as a vector in $\mathbb{F}_2^{128}$) is always 1. This assumption is needed to support point-and-permute.

only know the (masked-and-revealed) label $a = \lambda \oplus xR$ (resp. $b = \mu \oplus yR$). Note that the mask $\lambda$ (resp., $\mu$) corresponds to the label of the bit 0 in the classical free-XOR scheme.

Computing $c := a \oplus b$ yields the label of the plaintext bit $x \oplus y$ under mask $\nu = \lambda \oplus \mu$ (free-XOR property). For each AND gate ($x$ `AND` $y$), we use two efficient algorithms: `garble` (executed by the dealer/garbler) and `eval` (executed by the players/evaluator) whose signatures are the following:

$$\texttt{garble} : (\lambda, \mu) \in (\mathbb{F}_2^{128})^2 \mapsto (\nu, h_1, h_2) \in (\mathbb{F}_2^{128})^3 \tag{2}$$

$$\texttt{eval} : (a, b, h_1, h_2) \in (\mathbb{F}_2^{128})^4 \mapsto c \in (\mathbb{F}_2^{128}). \tag{3}$$

They need to satisfy the following properties: 1) $\forall x, y \in \{0, 1\}$, $\lambda, \mu \in \mathbb{F}_2^{128}$, and $(\nu, h_1, h_2) = \texttt{garble}(\lambda, \mu)$, one has $\texttt{eval}(\lambda \oplus xR, \mu \oplus yR, h_1, h_2) = \nu \oplus xyR$ (correctness); 2) knowing the labels $a = \lambda + xR$, $b = \mu + yR$ and the ciphertexts $h_1, h_2$ does not reveal any information about $\lambda, \mu, \nu$ or $R$ (privacy). Here, $h_1$ and $h_2$ are published and $\nu$ is held by the dealer. Given the hashes and the two labels $a, b$, the players can compute the label $c = \nu \oplus xyR$ without any communication. Constructing a correct `garble` and `eval` satisfying only 1) would be trivial - *i.e.*, take $\texttt{garble}(\lambda, \mu) = (0, \lambda, \mu)$ and $\texttt{eval}(a, b, h_1, h_2) = (a \oplus h_1)$ `AND` $(b \oplus h_2)$; however making it satisfy 1) and 2) simultaneously is hard: we refer to the classical half-gate technique [44] for a possible construction and its security proof. In this scheme, $h_1$ and $h_2$ correspond to the two ciphertexts generated by the dealer and published in the garbled table. Note that if one of inputs is known to the dealer or the player/evaluator, we do not need $h_2$. The protocol relies on a constant number of AES (native instructions) with the same secret key, and the ciphertexts are presented in the evaluation order in the garbled table, in order to improve memory latency. On GCP `c2-standard-8` instance, we are able to execute up to 10 millions of gates per second per core of players, e.g. an absolute value on a vector of 40000 fixed-point values per second per core per player, including the required pre-un-masking and post-re-masking for the A-to-Y and Y-to-A conversions.

In the classical garbled circuits scheme, labels are communicated via oblivious transfer from the garbler to the evaluator. In our context, the garbler is identified with the dealer, hence, is not present in the online phase. Thus, one uses secret sharing of both the key $R$ and the mask $\lambda$ instead of oblivious transfer. The players know all the input bits in plaintext (e.g. $x$ come from the bit-decomposition of modreal masked values), so they can jointly reveal the label $[\![\lambda \oplus xR]\!]_{\oplus}$ for each input bit $x$ of the circuit in a single initial round of communication. Then, the evaluation of the garbled circuit reveals each output (masked) bit. The advantage of this approach (the garbler being the dealer and the evaluator being the set of players) is that the evaluation of the whole circuit can be load-balanced across all the players in the sense that different players can evaluate different independent gates. This is not the case in arithmetic or Boolean shares where evaluating addition and multiplication yields a total amount of work and communication proportional to the number of players. Here, each gate is evaluated by only one of the players and the garbled table is split and distributed accordingly. The

amount of work per player to evaluate the whole circuit is inversely-proportional to the number of players $n$. This yields more flexibility in the trade-off between arithmetic shares, Boolean shares or garbled circuits where not only the complexity of the problem is essential, but also the number of players.

## 4 MPC Operations

In this section we describe our methods for MPC evaluation of the basic arithmetic operations as real-valued polynomials, division, exponential, logarithm and comparison. For the evaluation of linear combination and multiplication, we use classical approaches based on Beaver Triples [5] (described in the Supplementary Material Section 6).

### 4.1 Real-valued classical and trigonometric polynomials

The combination of lifts, additions and Beaver multiplications over plaintext classes and `Modreal` classes allows us to evaluate real-valued polynomials as well as trigonometric polynomials. Let

$$P(x) = \sum_{n=0}^{L} a_n x^n \text{ where } x \in [-1, 1] \text{ and } a_n \in \mathbb{R} \tag{4}$$

$$Q(x) = \mathrm{Re}\left(\sum_{n=0}^{L} c_n \cdot \exp\left(\frac{2i\pi n x}{T}\right)\right) \text{ where } x \in \mathbb{R}, c_n \in \mathbb{R} \text{ and } T \in \mathbb{R}. \tag{5}$$

Real-valued polynomials are first translated and rescaled in order to keep the variable $x$ in the interval $[-1, 1] \subseteq \mathbb{R}$. All coefficients are approximated at compile time with integers divided by the same power of 2: the plaintext and `Modreal` exponents $p_{\mathrm{lsb}}, p_{\mathrm{msb}}, M_{\mathrm{msb}}$ of the value $P(x)$ is therefore the `Modreal` exponents of the coefficients of $P$. Then, the secret shares of all the successive powers $(x, x^2, \ldots, x^L)$ are evaluated over `Modreal` shares within $\log_2 L$ rounds of lift and Beaver multiplications.

We adapt the approach of [9] to the case of evaluating trigonometric polynomials on `Modreal` secret-shared values with information-theoretic as opposed to statistical masking. More precisely, we first compute `Modreal` shares of $[\![x/T \mod 1]\!]$ (a direct application of the modular lifts from Section 3.3); we then use a uniformly random dealer-generated mask $\lambda \in [0, 1)$ to mask-and-reveal $x/T$, i.e., reveal $a = x/T + \lambda \mod 1$ to all the players. Finally,

$$[\![Q(x)]\!] = \mathrm{Re}\left(\sum_{n=0}^{L} [\![c_n \cdot \exp\left(-2i\pi n\lambda\right)]\!] . \exp\left(2i\pi n a\right)\right),$$

i.e., $Q(x)$ is a linear combination of pre-shared dealer-generated terms that only depend on $\lambda$. In other words, $Q(x)$ is evaluated within a single round of communication. Real polynomials are particularly adapted to functions whose Taylor

series has convergence radius $\geq 1$, whereas trigonometric polynomials are better suited for evaluation of functions that are either periodic or very close to a continuous periodic function on a segment of $\mathbb{R}$.

### 4.2 Oblivious comparisons

Comparison between two secret-shared values is performed using a combination of `Modreal`, `Boolean` operations or `Garbled circuits`. Roughly speaking, the bit sign of the difference $a - b$ is 1 when $a < b$ and 0 otherwise. Other comparisons (smaller or equal, greater and greater or equal) are computed equivalently either by switching the operands or by negating the outputted sign. The comparison method implemented in `Manticore` is not novel and is similar to the one implemented in [17]. In a more recent work [31] the oblivious comparison is further optimized, although a significant performance increase is observed only for comparisons over fields. Computing the sign bit requires a binary adder. Two binary adders are implemented: the ripple-carry (low multiplicative size) and the Sklansky adder [22] (low multiplicative depth). When using the `Boolean` backend the adder choice should be done as a function of the size of vectors to compare, the network (latency and bandwidth) and computation resources. In the context of `Garbled circuits`, the ripple-carry adder is the best choice.

**Benchmarks**: Unless specified otherwise, all our benchmarks run on machines with $4 \times 3.1$GHz cores, equipped with 32GB RAM (GCP `c2-standard-8` instances). A single processing core is used (no parallel execution). Two players are used. We log RAM usage, CPU time and end-to-end execution time of online/offline phases. Two types of network sizes are used: (i) the communication between the dealer and each of the players (the "triples" column) and (ii) the communication between the players during the online phase (the "network" column). End-to-end execution time includes communications which use a network infrastructure with bandwidth 120MBps and latency 0.3ms.

Table 2 gives benchmark results for different sizes of input vectors. Three implementations of the comparison are used: first two (Sklansky and ripple-carry adder) use the `Boolean` backend and the last one uses `Garbled circuits`. Input vectors are 60-bit wide. Being slower than the `Boolean` approaches, the `Garbled circuit` was included here for performance illustration purposes only. `Garbled circuits` having low online communication overhead, depending only on the input/output sizes and not on the executed circuit size, are not well suited for the comparison application.

### 4.3 Division, exponential and logarithm

**Division.** Given a `Modreal` number $x$, we approximate $1/x$ with an algorithm inspired by Goldschmidt's method (see [20] or [10]). However, unlike previous approaches, we allow higher order polynomials, yielding faster convergence, as explained below.

| Used implem. | Input size $\times 10^6$ | RAM | | Communication | | CPU | | End-to-end | |
|---|---|---|---|---|---|---|---|---|---|
| | | Offline | Online | Triples | Network | Offline | Online | Offline | Online |
| | | MB | | | | seconds | | | |
| `Boolean` Sklansky | 0.5 | 134 | 134 | 22 | 22 | 0.4 | 0.3 | 0.4 | 0.4 |
| | 5 | 277 | 483 | 216 | 215 | 4.6 | 3.3 | 4.6 | 4.4 |
| `Boolean` Ripple-carry | 0.5 | 131 | 132 | 17 | 18 | 0.4 | 0.3 | 0.4 | 0.4 |
| | 5 | 278 | 483 | 167 | 185 | 3.5 | 2.5 | 3.5 | 3.5 |
| Garbled circuit | 0.5 | 816 | 832 | 546 | 275 | 41.6 | 10.3 | 41.6 | 10.6 |
| | 5 | 7953 | 8046 | 5460 | 2747 | 415.4 | 103.7 | 415.4 | 105.8 |

**Table 2.** Private compare execution times and communication sizes.

The aim is to construct a sequence $\{w_i\}_{i\geq 0}$ of `Modreal` numbers that successively reduce the relative error

$$\varepsilon_i = \frac{1/x - w_i}{1/x} = 1 - xw_i$$

of approximating $1/x$ by $w_i$. In the sequel we assume $x > 0$.

We start by initialising the sequence with $w_0 = 2^{-\lfloor \log_2(x) \rfloor}$, computed via a binary circuit on the Boolean shares representation or the garbled circuit representation described in Section 3.4. Note that $1 \leq xw_0 < 2$.

In a first iteration step we compute the truncated Chebyshev series $c_{L_1}(z)$ (of certain degree $L_1$) of the function $1/z$ on the interval $[1, 2]$, and update

$$w_1 = w_0 \cdot c_{L_1}(xw_0).$$

If $\delta_0 = 1/xw_0 - c_{L_1}(xw_0)$ is the error of the approximation, it follows that $\varepsilon_1 = 1 - xw_1 = xw_0\delta_0$. Moreover, $\|1/z - c_{L_1}(z)\|_\infty \leq \frac{(3-2\sqrt{2})^{L_1+1}}{2-\sqrt{2}}$ (the norm being with respect to the interval $[1, 2]$), hence we deduce

$$|\varepsilon_1| \leq 2 \cdot \frac{(3-2\sqrt{2})^{L_1+1}}{2-\sqrt{2}}.$$

The elements $\{w_i\}_{i\geq 2}$, are computed via successive Taylor approximations of the function $1/z$ around 1.

**Lemma 1.** *Let $t_{L_i}(z) = \sum_{n=0}^{L_i}(1-z)^n$ be the truncated Taylor series of $1/z$ of degree $L_i$ around 1. If $w_i$ approximates $1/x$ with relative error $\varepsilon_i$, then*

$$w_{i+1} = w_i \cdot t_{L_i}(xw_i)$$

*approximates $1/x$ with relative error $\varepsilon_i^{L_i+1}$.*

*Proof.* This easily follows from $t_{L_i}(xw_i) = \sum_{n=0}^{L_i} \varepsilon_i^n = \frac{1-\varepsilon_i^{L_i+1}}{1-\varepsilon_i}$.

Note that in the prior works [20] and [10], the degree $L_i$ equals 1 throughout all iterations. In our approach we are free to vary the degree $L_i$ from iteration to iteration in order to find the best trade-off between computational cost and convergence rate. E.g. with a single Chebyshev approximation of degree 20 the relative error satisfies $|\varepsilon_1| \leq 2.862 \cdot 10^{-16}$. On the other side, with five iterations of degree 1 we have $|\varepsilon_1| \leq 1.005 \cdot 10^{-1}$, hence $|\varepsilon_5| = |\varepsilon_1|^{2^4} \leq 1.083 \cdot 10^{-16}$. We have implemented the variant with three iterations of degree 2 (requires least multiplications). Hence, $|\varepsilon_1| \leq 1.724 \cdot 10^{-2}$, and $|\varepsilon_3| = |\varepsilon_1|^{3^2} \leq 1.348 \cdot 10^{-16}$.

Table 3 shows execution times for different sizes of input vectors using the `Boolean` backend. Input vectors are 60-bit wide and the output has a relative precision of 50 bits compared to the plaintext division performed on double-precision floating-point.

| | | RAM | | Communication | | CPU | | End-to-end | |
|---|---|---|---|---|---|---|---|---|---|
| Used implem. | Input size $\times 10^6$ | Offline | Online | Triples | Network | Offline | Online | Offline | Online |
| | | MB | | | | seconds | | | |
| depth optimized | 0.5 | 176 | 302 | 331 | 261 | 4.6 | 3.8 | 4.6 | 5.2 |
| | 5 | 848 | 2822 | 3310 | 2611 | 50.7 | 42.8 | 50.7 | 56.0 |
| size optimized | 0.5 | 192 | 298 | 313 | 250 | 4.3 | 3.5 | 4.3 | 4.9 |
| | 5 | 808 | 2737 | 3133 | 2500 | 47.2 | 40.2 | 47.2 | 53.0 |

**Table 3.** Private division execution times and communication sizes.

**Exponential.** Similar to [3] our approach is based on the identity

$$2^x = 2^{\lfloor x \rfloor} \cdot 2^{x - \lfloor x \rfloor}.$$

However, we do not need to compute the absolute value of $x$ first, $2^{\lfloor x \rfloor}$ is computed via a binary circuit from the bit-decomposition of $x$ (as opposed to $p_{\text{msb}}^{(x)}$ multiplications in [3]), and $2^{x - \lfloor x \rfloor}$ is computed with a Chebyshev approximation of degree 10 on the interval $[0, 1]$ (as opposed to a Padé approximation of degree 8[6] in [3]), yielding 15 decimal digits of relative precision. Moreover, since we do not need to distinguish $x \geq 0$ from $x < 0$, we do not need to perform an obliviously selection between $2^x$ and $1/2^x$, hence saving a private division.

**Logarithm.** Assuming $x > 0$, similar to [3] our approach is based on the identity

$$\log_2(x) = w_0 + \log_2(x \cdot 2^{-w_0}),$$

where $w_0 = \lfloor \log_2(x) \rfloor$ (both $w_0$ and $2^{-w_0}$ are computed via a binary circuit from the bit-decomposition of $x$). Analogous to the division algorithm presented

[6] With the polynomial $P_{1045}$ from the appendix of [3] we observed a relative error $|1 - P_{1045}(x)/2^x|$, $x \in [0, 1]$, of roughly $10^{-7.11}$ as opposed to the $10^{-12.11}$ claimed in the paper.

above, we want to construct a sequence $\{w_i\}_{i\geq 1}$ of `Modreal` numbers that successively reduce the error

$$\epsilon_i = \log_2(x) - w_i.$$

At each iteration, $w_{i+1}$ is obtained by computing an approximation, say $a_i$, of certain degree $L_i$ of $\log_2(x \cdot 2^{-w_i})$ and updating

$$w_{i+1} = w_i + a_i.$$

Observe that $x \cdot 2^{-w_i}$ converges towards 1, since $x \cdot 2^{-w_i} = 2^{\epsilon_i}$.

Computing the logarithm through iterative steps allows one to optimise a similar convergence rate versus computation cost trade-off as for the division, while previous methods only allow one iteration (e.g. [3] with one Padé approximation[7]).

In a first iteration step we compute the truncated Chebyshev series $c_{L_1}(z)$ (of certain degree $L_1$) of the function $\log_2(z)$ on the interval $[1, 2]$, and update $w_1 = w_0 + a_1$, where $a_1 = c_{L_1}(x \cdot 2^{-w_0})$. One can show that

$$\| \log_2(z) - c_{L_1}(z)\|_\infty \leq \frac{(3 - 2\sqrt{2})^{L_1+1}}{(L_1 + 1)\log(2)(\sqrt{2} - 1)}$$

(the norm being with respect to the interval $[1, 2]$), which gives a bound for $|\epsilon_1|$. In the next iteration step we need to approximate $\log_2(x \cdot 2^{-w_1})$, therefore we need to compute $x \cdot 2^{-w_1} = (x \cdot 2^{-w_0}) \cdot 2^{-a_1}$. Since $0 \leq \log_2(x \cdot 2^{-w_0}) < 1$ we compute $2^{-a_1}$ with a Chebyshev approximation of the function $2^z$ on the interval $[-1, 0]$.

The elements $\{w_i\}_{i\geq 2}$ are computed via successive Taylor approximations of the function $\log_2(z)$ around 1.

**Lemma 2.** *Let $t_{L_i}(z) = \frac{-1}{\log(2)} \sum_{n=1}^{L_i} \frac{(1-z)^n}{n}$ be the truncated Taylor series of $\log_2(z)$ of degree $L_i$ around 1. If we update $w_{i+1} = w_i + a_i$, where $a_i = t_{L_i}(x \cdot 2^{-w_i})$, then the error $\epsilon_{i+1}$ satisfies*

$$|\epsilon_{i+1}| \leq \frac{|\epsilon_i|^{L_i+1}}{(L_i + 1)\log(2)(1 - |\epsilon_i|)}.$$

*Proof.* One can easily see that

$$\begin{aligned} \epsilon_{i+1} &= \log_2(x \cdot 2^{-w_i}) - t_{L_i}(x \cdot 2^{-w_i}) \\ &= \frac{-1}{\log(2)} \sum_{n \geq L_i+1} \frac{(1 - 2^{\epsilon_i})^n}{n}, \end{aligned}$$

i.e. is equal to the error of the Taylor approximation. The proof follows from the fact that $|1 - 2^\epsilon| \leq |\epsilon|$ in a neighbourhood of 0.

[7] Numerator and denominator are of degree 3, yielding a relative error of $10^{-8.32}$.

For the upcoming iteration we need to compute $x \cdot 2^{-w_{i+1}} = (x \cdot 2^{-w_i}) \cdot 2^{-a_i}$. We compute $2^{-a_i}$ with a Taylor approximation of the function $2^z$ around 0 ($a_i = \epsilon_i - \epsilon_{i+1}$ is close to 0).

Since the update rule from one iteration to the next one is considerably more expensive than for the division (as it amounts to compute $2^{-a_i}$), it is preferable to keep the number of iterations low. The variant we have implemented is: degree-8 Chebyshev approximation followed by one degree-1 Taylor approximation, and yields $|\epsilon_1| \leq 4.985 \cdot 10^{-8}$, and $|\epsilon_2| \leq 1.793 \cdot 10^{-15}$. As a comparison, a degree-18 Chebyshev approximation yields $|\epsilon_1| \leq 5.220 \cdot 10^{-16}$.

## 5 Applications

### 5.1 Logistic regressions and principal component analysis

Recall that, given a binary vector $y$ and a feature matrix $X$ (of size $N \times k$), the cost function of a particular model $\theta \in \mathbb{R}^k$ is

$$\mathcal{L}(X, y, \theta) = -\sum_{i=1}^{N} \Big( y_i \log(\sigma(X^{(i)}\theta)) + (1 - y_i) \log(1 - \sigma(X^{(i)}\theta)) \Big),$$

where $X^{(i)}$ denotes the $i$th row of $X$ and $\sigma(x) = 1/(1 + e^{-x})$ is the sigmoid function. For fixed $X$ and $y$ and varying $\theta$, the function $\mathcal{L}$ is convex so it has a unique minimum. Furthermore, the $L2$-regularized cost function is given by (where, $\lambda$ is the regularization parameter):

$$\mathcal{L}_{\text{reg}}(X, y, \theta, \lambda) = \mathcal{L}(X, y, \theta) + \lambda \sum_{i=1}^{k} \theta_i^2.$$

To find the minimum, we use several (relatively cheap) gradient descent (first-order) steps followed by several more expensive IRLS/Newton–Raphson (second order) steps [43]. The intuition is that gradient descent still improves the model a lot in the first few iterations and then we will switch to the more expensive IRLS iterations for faster convergence.

*Principal component analysis (PCA).* In order to reduce the complexity of the regression algorithms and obtain better precision on the final model we can apply a singular value decomposition (SVD) on the dataset $X$ and work on the (normalized) principal components instead. The benefits of the PCA are two-fold:

(i) it removes singularities and reduces the dimension of the feature space of the data to a subspace where the data has high variance,
(ii) the change of variables allows us to work on a dataset with orthonormal[8] columns, which significantly increases the performance and stability of the algorithm.

[8] A set of vectors is orthonormal if every vector in the set has norm 1 and the set of vectors are mutually orthogonal.

Let $r$ be the rank of $X$. Our convention for the singular value decomposition is $X = U\Sigma V^t$, where $\Sigma$ is a $r \times r$ diagonal matrix whose diagonal entries are the non-zero singular values of $X$ in decreasing order, $U$ is a $N \times r$ matrix whose columns are the first $r$ normalized principal components and $V$ is a $k \times r$ matrix whose columns are the first $r$ principal axes. Note that both $U$ and $V$ have orthonormal columns. We have two options to compute the SVD of $X$ (either way we obtain the singular value decomposition by first computing the eigendecomposition of $X^tX$). In the first option, we mask $X^tX$ with a random secret orthogonal matrix $P$ (generated by the trusted dealer) and compute a public eigendecomposition on the revealed matrix $P^tX^tXP$. This reveals the singular values $\Sigma$. In the second option, we compute a private eigendecomposition using a garbled circuit as in [1]. We then need to take private square-roots in order to obtain $\Sigma$ from the spectrum of $X^tX$. No additional information is revealed in this approach. In our implementation, the default option is the first one.

Since $X$ and $U$ span the same subspace of $\mathbb{R}^N$, for every model $\theta \in \mathbb{R}^k$, the prediction $\widehat{y} = X \cdot \theta$ can be obtained as $\widehat{y} = U \cdot \theta_{\text{red}}$ for a unique $\theta_{\text{red}} \in \mathbb{R}^r$. That is, instead of minimizing $\mathcal{L}(X, y, \theta)$ one would rather minimize

$$\mathcal{L}_{\text{red}}(U, y, \theta_{\text{red}}) = -\sum_{i=1}^{N} \Big( y_i \log(\sigma(U^{(i)}\theta_{\text{red}})) + (1 - y_i) \log(1 - \sigma(U^{(i)}\theta_{\text{red}})) \Big) .$$

If $\theta_{\text{red}}$ minimizes $\mathcal{L}_{\text{red}}(U, y, \theta_{\text{red}})$ then $\theta := V\Sigma^{-1}\theta_{\text{red}}$ minimizes $\mathcal{L}(X, y, \theta)$. Moreover, $\theta$ is the model of least $L2$-norm among all models that minimize $\mathcal{L}(X, y, \theta)$. This follows from the fact that $V$ and $\ker(X)$ form an orthogonal decomposition of $\mathbb{R}^k$, i.e. $V\Sigma^{-1}\theta_{\text{red}}$ is the unique minimum of $\mathcal{L}(X, y, \theta)$ that is orthogonal to $\ker(X)$.

*Remark 1.* We can also improve the precision by doing a pre-PCA (rank-detection) step. In this case, the user must provide a PCA rank (or the rank $r$ of the matrix if he wants to keep all the dataset), and the order of magnitude of the smallest remaining singular value (in general, a very small 6-bit integer between $-32$ and 32). By passing this information (and the true rank if we do not do a PCA), we go from 5 to 10 precision digits: a.k.a. same precision or even better than `sklearn`).

*Internal normalization.* In case $X$ has columns of completely different orders of magnitude, the singular value decomposition of $X$ cannot be computed with sufficient precision and hence, the PCA method does not yield the desired result. In this case we need to normalize the columns of $X$ first. Letting $\Delta$ be a $k \times k$ diagonal matrix with entries being bounds on the respective column of $X$ (e.g. the $L1$-, $L2$- or $L_\infty$-norm), we define $X_n = X\Delta^{-1}$ the *normalized* dataset. Note that in our implementation we choose $\Delta$ with $\Delta_{jj} = 2^{\lceil \log_2(\|X_{(j)}\|_\infty) \rceil}$, i.e. the closest power of 2 that upper-bounds the maximum of the column. This is either computed and revealed prior to the MPC computation (and $X$ is normalized via a public-private multiplication) or it is computed during the MPC computation (and $X$ is normalized via a private division, as explained in Section 4.3). The

two matrices $X$ and $X_n$ span the same subspace of $\mathbb{R}^N$, hence one can minimize $\mathcal{L}(X_n, y, \theta)$ instead. Similar to the above paragraph we compute a singular value decomposition $X_n = U_n \Sigma_n V_n^t$. One then minimizes the convex function $\mathcal{L}_{\text{red}}(U_n, y, \theta_{\text{red}})$ and returns $M \cdot \theta_{\text{red}}$, where $M \in \text{Mat}_{k \times r}(\mathbb{R})$ sends the minimum of $\mathcal{L}_{\text{red}}(U_n, y, \theta_{\text{red}})$ to the unique minimum of $\mathcal{L}(X, y, \theta)$ that is orthogonal to $\ker(X)$.

*Evaluating the sigmoid function.* We have two approaches to evaluate the sigmoid function $\sigma(x)$. Either by the direct formula $\sigma(x) - 1/2 = \text{sign}(x)(1/(1 + e^{-|x|}) - 1/2)$, where sign and absolute value are based on Boolean circuits, and exponential and division are from Section 4.3, or we approximate the centered sigmoid on a "large enough" interval $[-B, B]$ with the $4B$-periodic function $-\frac{1}{2} + \sum_{k \in \mathbb{Z}} \sigma(x + 4kB) - \sigma(x + (4k + 2)B)$, whose Fourier coefficients decay exponentially fast. In order to obtain accurate sigmoid evaluations outside the Fourier interpolation interval, we use Boolean comparison circuits to detect if the input is outside the interpolation interval $[-B, B]$. Afterwards, the resulting binary values are used to choose between the approximation when $x \in [-B, B]$, $-1/2$ when $x < -B$ and $1/2$ when $x > B$.

Our benchmarks are presented with the Fourier approximation approach, which is faster when the desired precision is around 28 bits, which is enough for logistic regression use-cases. If one desires higher precision (e.g. more than 50bits), the increasing degree of the approximation and the lack of hardware support of trigonometric functions make the direct computation of the sigmoid preferable.

*Blockwise optimizations.* In order to improve run-time and memory for large-sized feature matrices, we use blockwise approach: matrix products with large inputs and small outputs can be computed blockwise. An example is the Hessian [42] of $\mathcal{L}$, which is a $k \times k$ matrix $W^t W$, where $W$ is a $N \times k$ matrix with $N \gg k$, and where $W$ can be divided vertically into smaller row-blocks. This reduces the memory footprint for the same running time and bytes of communication. Another improvement can be applied to any convex gradient descent problem: classical and IRLS gradient descent share a common update formula where at each iteration $\theta \leftarrow \theta - H^{-1}\nabla$. In the IRLS case, $H$ is the Hessian matrix of the cost function $\mathcal{L}$. In the classical case, $H$ is a constant diagonal matrix containing the inverse learning rate, which must be of the order of magnitude of the Hessian diagonal coefficients. In other advanced gradient descent (with adaptive learning rate), $H$ contains only the diagonal coefficients of the Hessian of $\mathcal{L}$, and 0 everywhere else. As a simple rule of thumb, the more $H$ coincides with the Hessian, the less iterations are required to reach full convergence. Computing $p$ coefficient out of $k(k + 1)/2$ yields a time speed-up of roughly $p/k^2$ on the computation of $H$. Note that independently of how many coefficients we compute, one logreg iteration cost $O(Nk)$ bytes of RAM and communication. In our experiments, if the number of features $k$ is small enough so that we can afford a $O(nk^2)$ running time, it is preferable to compute the full Hessian at each iteration. In theory, within the domain of convergence, Newton method squares at

each iteration the number fractional bits that have converged. Experimentally, we verified that between 8 and 12 IRLS iterations are sufficient for logreg to converges with 50 bits of precision for all orthonormal datasets, even with more than 500 features. If $k$ is too large to afford cubic running time, the trade-off we propose consists in computing only diagonal blocks of fixed size $B$ in $H$, and leave the rest of the coefficients zero. The complexity of computing $H$ drops to $O(Nk)$ (see Fig. 2), inversion of $H$ requires to invert only fixed dimension $B \times B$ blocks, and most importantly, the total number of iterations still remains much lower than in the classical gradient descent case.

**Fig. 2.** In left we represent the number of coefficients to compute in case of gradient descent, in middle the full Hessian matrix and in right our block-wise approach. Note that just the elements in grey are computed.

**Benchmarks.** Table 4 shows computation overhead for datasets with 50 features and different number of rows. Synthetic instances are randomly generated. For the last 2 instances we have used higher memory machines (400GB of RAM for the last one!). In the Supplementary Material 6 benchmark results are given for larger number of players (3 and 5) and different number of features (10, 50, and 100).

| | RAM | | Communication | | CPU | | End-to-end | |
|---|---|---|---|---|---|---|---|---|
| Input size | Offline | Online | Triples | Network | Offline | Online | Offline | Online |
| $\times 10^6$ | GB | | | | minutes | | | |
| 0.5 | 2.3 | 3.5 | 14.1 | 16.6 | 6.3 | 5.5 | 6.3 | 6.7 |
| 5 | 26.5 | 45.0 | 150.8 | 130.3 | 105.6 | 46.4 | 105.6 | 55.7 |
| 50 | 263.8 | 381.4 | 1408.8 | 1664.1 | 689.8 | 780.6 | 689.8 | 861.2 |

**Table 4.** Logreg execution times and communication sizes.

We then have tested the *convergence* of our MPC logistic regression algorithm against the plaintext implementation of `scikit-learn`[9], by looking at the loss difference $\mathcal{L}_{\text{diff}}$ between the trained MPC model ($\theta_{\text{mpc}}$) and plaintext model ($\theta_{\text{skl}}$), $\mathcal{L}_{\text{diff}} = \mathcal{L}(X, y, \theta_{\text{mpc}}) - \mathcal{L}(X, y, \theta_{\text{skl}})$. We have generated numerous

[9] https://scikit-learn.org

classification datasets $(X, y)$ by varying the dimension $(N \times k)$ of $X$ and the *magnitude* of $X$. The scale $s_j$ of a column $j$ is defined as the smallest integer such that the inequality $\max(|X_j|) < 10^{s_j}$ hold.

The scale range $S$ of a matrix $X$ is $\max_j s_j - \min_j s_j$. The numerical stability of the logistic regression decreases with the increase of the scale $S$, because the fixed-point implementation has a lower precision.

We have then computed the worst loss difference $\mathcal{L}_{\text{diff}}$ over 5 runs for different values of $\lambda$ (regularization parameter), $\min_j s_j$ (`X scale min`), and $\max_j s_j$ (`X scale max` axis). Partial results are shown in Figure 3 for $N = 10000, k = 30, \text{rank}(X) = 20$. Each of the 10 singular columns was randomly generated from a subset of the independent columns of $X$. For readability $\mathcal{L}_{\text{diff}}$ is truncated to a power of 10 in the heatmap, and values $< 10^{-15}$ are capped at $10^{-15}$ (i.e. $\approx 0$). Observe that for a scale range $S \leq 6$, the light blue region in the chart, `Manticore` logistic regression converges to the same loss than `scikit-learn`, i.e $\mathcal{L}_{\text{diff}} \approx 0$. When $6 < S < 14$ (dark blue region), we achieve a better loss than `scikit-learn`, thanks to the internal *normalization* strategy. When $S \geq 14$, our algorithm does not converge to the minimal cost model due to overshooting of the rank estimate.

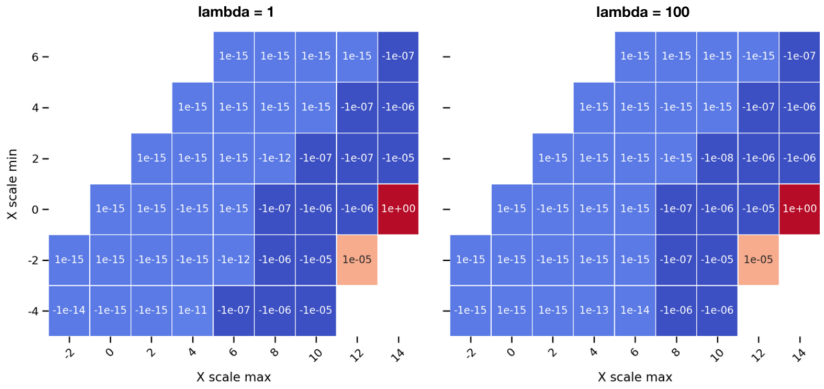



**Fig. 3.** Log-loss difference ($\mathcal{L}_{\text{diff}}$) Manticore vs scikit-learn

We shall note that most of MPC implementations of logistic regression require full rank and clean input data, and would converge only in the cell $(0, 0)$ (in the context of Figure 3). In Supplementary Material 6, we provide more benchmarks on the numerical precision of `Manticore` versus `MP-SPDZ` [25] versus the plaintext implementation of `scikit-learn`.

### 5.2 An oblivious sorting algorithm based on quicksort

One of the major challenges with efficient oblivious sorting algorithms in the MPC setting has been the fact that the two most efficient sorting algorithms,

merge-sort and quicksort, are data dependent. Efforts have been made to design oblivious versions of these two algorithms [23] (implementing Batcher's merge sort on Sharemind's system [6]), [21], [13] (the latter combining [6], [21] and Batcher sorting networks).

Here, we discuss a quicksort variant implemented on the `Manticore` framework. Using an idea that already appears in prior work (e.g., [21], [13]), we first make all elements unique by appending a least significant counter tag of $\log_2(N)$ bits to the elements of the original (secret-shared) array, then we *shuffle* the array by applying a dealer-generated secret-shared random permutation. We then perform (obliviously) the comparisons of the quicksort algorithm on the shuffled array and reveal the Boolean outputs at each iteration. The uniformly random permutation ensures that no information is leaked about the relative order of the original array.

Shuffling the original vector represents only a minor amount of time in quicksort. To apply the dealer-generated permutation, we choose between two implementations: if the number of players is large ($k \geq \log_2(N)$) we use a Benes network [41] [11], where the dealer picks a uniformly random permutation, routes it, and secret-share the Boolean network switches. This enables parallelizable shuffling (using the network) within exactly $2 \cdot \log(N)$ rounds of hadamard products of size $N$. In the opposite, when the number of players is small ($k \leq \log_2(N)$), we use a simpler protocol, inspired by [13], where the dealer sends one permutation to each player, and during the online phase, the players simply evaluate the composition of these permutations, in exactly $k$ rounds of secret permutation. Another key idea for our implementation is the concept of multi-pivots: recall that quicksort is a recursive procedure that, in its most basic form, chooses an element of the array (a pivot) and first permutes the elements according to the pivot (the smaller ones being on the left and the larger ones being on the right of the pivot), thus partitioning the array into two subarrays. It then calls itself on the two subarrays. The worst case of the algorithm thus occurs for highly unbalanced partitions (one of the sets has much more elements than the other one). The random permutation applied in the beginning is addressing this issue on average. In order to further reduce the variance on the expected depth of the algorithm, we partition the array into more than two sets by using more than one pivot and use oblivious comparisons in parallel. [4] Unlike plaintext sorting where sequential comparisons do not become the bottleneck, the more expensive oblivious comparisons would benefit from batching/parallelization, thus, making multiple pivots particularly useful. Since the number of rounds of communication is proportional to the depth, the multi-pivot approach provides a heuristically faster algorithm.

Overall, the algorithm runs in $\mathcal{O}(b \cdot \log_2 N)$ communication rounds where $b$ is the bit-size of each element of the array and $N$ is the number of elements. The Benes network requires $2 \cdot \log_2 N$ communication rounds of complexity $N/2$ each whereas the composition protocol requires $k$ communication rounds of complexity $N$ each, where $k$ is the number of players. Moreover, quicksort runs in (heuristically) $\log_2 N$ comparison rounds.

We evaluate experimentally the quicksort algorithm in our common real world testbed with a trusted dealer and two players (c.f Table 5).

| | RAM | | Communication | | CPU | | End-to-end | |
|---|---|---|---|---|---|---|---|---|
| Input size $\times 10^6$ | Offline | Online | Triples | Network | Offline | Online | Offline | Online |
| | MB | | | | seconds | | | |
| 0.5 | 68 | 138 | 479 | 509 | 6.5 | 7.3 | 6.5 | 10.1 |
| 5 | 521 | 697 | 3649 | 5177 | 64.7 | 69.2 | 64.7 | 98.5 |
| 50 | 4893 | 4940 | 41175 | 59923 | 788.9 | 904.5 | 788.9 | 1223.0 |

**Table 5.** Benchmark results for `Manticore` Quicksort.

## 6 Security Analysis

**Threat model**. In our threat model, we assume that all communication channels are private and authenticated. Adversaries are impersonated with the computing parties abbreviated as CP's hereafter. Any security leakage that may occur after the reveal of the value of the function $f$ over the inputs data is beyond our security scope. Moreover, we assume that input parties provide the correct input to the protocol and that the TD does not participate in the online phase. Our security model is the full-threshold model, that is, no coalition of $n-1$ parties can learn even a single bit about the input data.

Before going into formal proofs, we highlight that the security of `Manticore` relies upon: 1) TD not participating in the online computations; 2) At most $n-1$ CP's colluding; 3) Uniformly random secret shares of each value. The required quality of randomness during the offline phase is guaranteed by the security properties of the underlying `AES-CTR-DRBG` design.

**A security theorem**. `Manticore` uses immutable MPC variables where values are distributed (shared) among the players. Each player has its own view of an MPC variable, it can be secret shared (SSVar), masked-and-revealed (MRVar), or revealed (RevVar). For each SSVar, the $i$th player has access to the secret share $x_i$ of a plaintext value $x$ and the secret share $\lambda_i$ of the plaintext mask $\lambda$. For each masked-and-revealed MPC variable MRVar, the $i$th player broadcasts the masked value $a_i = \lambda_i + x_i$ to all other players. Moreover, the $i$th player has access to the shares $x_i$ and $\lambda_i$. The reveal MPC variables RevVar are the values revealed to all players (i.e., intermediate or output values).

MPC programs are composed of builtins VBuiltIn (e.g. Beaver multiplications, linear combinations, private comparisons, etc.) where each builtin operates on MPC variables SSVar, MRVar, RevVar. In addition, builtins include ephemeral shares of masked values (e.g., shares of values such as $\lambda \cdot \mu$ in Beaver multiplication). These are shares that are not associated with a particular MPC variable - we denote the set of these shares by ES. We assume they are indexed by

ephemeral identifiers $\texttt{eid} \in \mathsf{ES}$. Finally, for each player $i$, we let $\mathsf{InVar}_i$ be the set of plaintext input values coming from that player. For a coalition $G$ of players, we let $\mathsf{InVar}(G) = \bigcup_{i \in G} \mathsf{InVar}_i$.

We first introduce the various random variables involved by distinguishing three higher-level types: for a coalition $G$ of at most $n-1$ corrupted parties, we define 1) input random variables $\texttt{IN}(G)$ as all input plaintext values available to the parties in $G$, 2) revealed random variables $\texttt{REV}$ as all the information explicitly revealed to all parties (this includes all the revealed outputs as well) and 3) the view random variables $\texttt{VIEW}(G)$ as all the information available to the coalition parties in $G$ throughout the protocol (including $\texttt{IN}(G)$, $\texttt{REV}$ as well as all the secret-shares available to $G$, all data transmitted over the network, all the masked-and-revealed values, etc.).

More precisely, for $i = 1, \ldots, n$, let $X_{\mathsf{id}}$, $X_{i,\mathsf{id}}$ and $\Lambda_{i,\mathsf{id}}$ denote the random variables corresponding to the plaintext value, player $i$'s secret shares of $X_{\mathsf{id}}$ and the plaintext mask for the MPC variable with identifier $\mathsf{id}$. Moreover, let $E_{i,\texttt{eid}}$ be the ephemeral secret share with ephemeral identifier $\texttt{eid}$. Note that these variables take values over the corresponding spaces of secret shares for the corresponding parameters or output values (e.g., a builtin might output a Boolean value or a real value). Since the parameters $(p_{\mathrm{msb}}, p_{\mathrm{lsb}})$ of each MPC variable are statically known (before the execution of the program), we know exactly the space of values of each of these random variables at compile time.

With these in mind, we can define (for the coalition $G$ introduced above)

$$\texttt{IN}(G) := \prod_{\mathsf{id} \in \mathsf{InVar}(G)} X_{\mathsf{id}}, \quad \texttt{REV} := \prod_{\mathsf{id} \in \mathsf{RevVar}} X_{\mathsf{id}},$$

$$\texttt{COL}(G) := \prod_{\substack{i \in G \\ \mathsf{id} \in \mathsf{SSVar} \cup \mathsf{MRVar}}} X_{i,\mathsf{id}} \times \prod_{\substack{i \in G \\ \mathsf{id} \in \mathsf{MRVar}}} \Lambda_{i,\mathsf{id}} \times \prod_{\substack{i \in G \\ \texttt{eid} \in \mathsf{ES}}} E_{i,\texttt{eid}},$$

$$\texttt{NET} := \prod_{\substack{i \in [1,n] \\ \mathsf{id} \in \mathsf{MRVar}}} (X_{i,\mathsf{id}} + \Lambda_{i,\mathsf{id}}) \times \prod_{\substack{i \in [1,n] \\ \mathsf{id} \in \mathsf{RevVar}}} X_{i,\mathsf{id}},$$

$$\texttt{VIEW}(G) := \texttt{COL}(G) \times \texttt{NET} \times \texttt{IN}(G) \times \texttt{REV}.$$

Finally, if $X$ is a random variable over the plaintext values (e.g., $X$ can correspond to the least significant bit or the most significant bit of a private input), we will be interested in the probabilities $\Pr_{\mathcal{A}}(X | \texttt{VIEW})$ and $\Pr_{\mathcal{A}}(X | \texttt{IN}(G) \wedge \texttt{REV})$. Knowing that these are equal or very similar would mean that the the extra information of knowing $\texttt{VIEW}$ would not help a coalition of $n-1$ players in guessing the value of $X$ (conditional on the input plaintext values $\texttt{IN}(G)$ and the output/revealed values $\texttt{REV}$).

In order to satisfy the above properties for the individual builtins of `Manticore`, we often need to randomize the output shares revealed to the user with fresh secret shares of zero. The reason is that there might be operations (such as $[\![x]\!] - [\![x]\!]$) where the resulting secret shares are $(0, \ldots, 0)$. Randomize all output

shares with new, independent secret shares of zero guarantees that the secret shares in the output containers are independent and uniformly random over the corresponding sets of definition.

We will state a general security theorem proving this equality under three facts that we will verify for the builtins of our particular protocol:

**Fact 1** *The network communication is exclusively limited to the random variables in* $\texttt{NET}$*, the rest of the computation in the protocol being local to each player.*

**Fact 2** *For any coalition* $G$ *of at most* $n-1$ *players the following property holds: for any* $i \in G$ *and any* $\mathsf{id} \in \mathsf{MRVar}$*,* $\lambda_{i,\mathsf{id}}$ *are independent and uniformly random, and are independent from* $x_{j,\mathsf{id}'}$ *for all* $j \in G$ *and* $\mathsf{id}' \in \mathsf{SSVar} \cup \mathsf{MRVar}$*, and all* $x_{\mathsf{id}'}$ *for* $\mathsf{id}' \in \mathsf{RevVar} \cup \mathsf{InVar}$*.*

**Fact 3** *For any coalition* $G$ *of at most* $n-1$ *players the following property holds: for any* $i \in G$*, any* $\mathsf{id} \in \mathsf{SSVar} \cup \mathsf{MRVar}$*, for all* $j = 1, \ldots, n$*, and any* $\mathsf{id}' \in \mathsf{RevVar}$ $x_{i,\mathsf{id}}$ *and* $x_{j,\mathsf{id}'}$ *are all independent and uniformly random, and they are independent from all* $x_{\mathsf{id}''}$ *for* $\mathsf{id}'' \in \mathsf{InVar}$*.*

All the builtins operate by construction under Fact 1, and draw each MPC variable's mask uniformly at random (to enable seeding), which implies Fact 2. Fact 3, if not already satisfied, is guaranteed by additional re-randomization of the output secret shares of each builtin.

**Theorem 1.** *Assume Facts 1, 2 and 3. Then for any coalition* $G$ *of at most* $n-1$ *players, any distribution on a (random) variable* $X$ *on the plaintext values and any subset values* $x$*,* $c_G$*,* $n$*,* $i_G$ *and* $r$ *for* $X$*,* $\texttt{NET}$*,* $\texttt{COL}(G)$*,* $\texttt{IN}(G)$ *and* $\texttt{REV}$*, respectively, we have*

$$\Pr_{\mathcal{A}}[X \in x | \, \texttt{VIEW}(G) \in v_G] = \Pr_{\mathcal{A}}[X \in x | \, \texttt{IN}(G) \in i_G, \ \texttt{REV} \in r],$$

*where* $v_G = c_G \times n \times i_G \times r$*.*

We now sketch a proof of Theorem 1:

*Proof.* (Sketch) Letting $A_{i,\mathsf{id}} := X_{i,\mathsf{id}} + \Lambda_{i,\mathsf{id}}$, $G^c$ to be the (non-empty) complement of $G$, and suppressing (for brevity) the value subsets for the random

variables when they are understood from the context, we can write:

$$
\begin{aligned}
P &= \Pr[X \mid \texttt{VIEW}(G)]) \\
&\overset{(1)}{=} \Pr\left[X \mid \bigwedge_{\mathsf{id}\in\mathsf{InVar}(G)} X_{\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{SSVar}\cup\mathsf{MRVar}}} X_{i,\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{MRVar}}} \Lambda_{i,\mathsf{id}} \wedge \bigwedge_{\substack{1\le i\le n \\ \mathsf{id}\in\mathsf{MRVar}}} A_{i,\mathsf{id}} \wedge \bigwedge_{\substack{1\le i\le n \\ \mathsf{id}\in\mathsf{RevVar}}} X_{i,\mathsf{id}} \wedge \bigwedge_{\mathsf{id}\in\mathsf{RevVar}} X_{\mathsf{id}}\right] \\
&\overset{(2)}{=} \Pr\left[X \mid \bigwedge_{\mathsf{id}\in\mathsf{InVar}(G)} X_{\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{SSVar}\cup\mathsf{MRVar}}} X_{i,\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{MRVar}}} \Lambda_{i,\mathsf{id}} \wedge \bigwedge_{\substack{1\le i\le n \\ \mathsf{id}\in\mathsf{MRVar}}} A_{i,\mathsf{id}} \wedge \bigwedge_{\mathsf{id}\in\mathsf{RevVar}} X_{\mathsf{id}}\right] \\
&\overset{(3)}{=} \Pr\left[X \mid \bigwedge_{\mathsf{id}\in\mathsf{InVar}(G)} X_{\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{SSVar}\cup\mathsf{MRVar}}} X_{i,\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{MRVar}}} \Lambda_{i,\mathsf{id}} \wedge \bigwedge_{\substack{j\in G^c \\ \mathsf{id}\in\mathsf{MRVar}}} A_{j,\mathsf{id}} \wedge \bigwedge_{\mathsf{id}\in\mathsf{RevVar}} X_{\mathsf{id}}\right] \\
&\overset{(4)}{=} \Pr\left[X \mid \bigwedge_{\mathsf{id}\in\mathsf{InVar}(G)} X_{\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{SSVar}\cup\mathsf{MRVar}}} X_{i,\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{MRVar}}} \Lambda_{i,\mathsf{id}} \wedge \bigwedge_{\mathsf{id}\in\mathsf{RevVar}} X_{\mathsf{id}}\right] \\
&\overset{(5)}{=} \Pr\left[X \mid \bigwedge_{\mathsf{id}\in\mathsf{InVar}(G)} X_{\mathsf{id}} \wedge \bigwedge_{\substack{i\in G \\ \mathsf{id}\in\mathsf{SSVar}\cup\mathsf{MRVar}}} X_{i,\mathsf{id}} \wedge \bigwedge_{\mathsf{id}\in\mathsf{RevVar}} X_{\mathsf{id}}\right] \\
&\overset{(6)}{=} \Pr\left[X \mid \bigwedge_{\mathsf{id}\in\mathsf{InVar}(G)} X_{\mathsf{id}} \wedge \bigwedge_{\mathsf{id}\in\mathsf{RevVar}} X_{\mathsf{id}}\right] \\
&= \Pr[X \mid \texttt{IN}(G) \wedge \texttt{REV}]
\end{aligned}
$$

- Implication (1): follows from Fact 1 as well as the fact that the shares $e_{i,\texttt{eid}}$ for $i \in G$ are uniformly random in $\mathcal{M}_{M_{\mathrm{msb}},p_{\mathrm{lsb}}}$ and independent from the rest of the random variables. As such, Bayes rule allows us to remove them from the conditional event.
- Implication (2): follows from Fact 3.
- Implication (3): we remove $a_{i,\mathsf{id}}$ for $i \in G$ since $a_{i,\mathsf{id}} = x_{i,\mathsf{id}} + \lambda_{i,\mathsf{id}}$ and $x_{i,\mathsf{id}}, \lambda_{i,\mathsf{id}}$ already exist in the formula form previous terms. As such, we only keep $a_{i,\mathsf{id}}$ for $i \in G^c$.
- Implication (4): $\lambda_{i,\mathsf{id}}$ for $i \in G^c$ are independent from $\lambda_{i,\mathsf{id}}$ for $i \le n$ and the rest of the random variables ($\lambda_{i,\mathsf{id}}$ for $i \in G$ act as one-time-pad encryption key), that makes $a_{i,\mathsf{id}} = \lambda_{i,\mathsf{id}} + x_{i,\mathsf{id}}$ independent for $i \in G^c$ and Bayes rule allows to remove them.
- Implication (5): follows from Fact 2.
- Implication (6): follows from Fact 3.

**Conclusion** We presented a novel MPC framework `Manticore` in the multiparty setting with a highly efficient and scalable implementation of the principal arithmetic operations as well a logistic regression training algorithm, robust against

singular features and unbalanced datasets, and an improved oblivious sorting. We provided the security proof of our framework in a semi-honest model with an offline dealer and full-threshold security across an arbitrary number of players.

## References


1. Al-Rubaie, M., Wu, P.y., Chang, J.M., Kung, S.Y.: Privacy-preserving pca on horizontally-partitioned data. In: 2017 IEEE Conference on Dependable and Secure Computing. pp. 280–287. IEEE (2017)
2. Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. Cryptology ePrint Archive, Report 2012/405 (2012), https://eprint.iacr.org/2012/405
3. Aly, A., Smart, N.P.: Benchmarking privacy preserving scientific operations. Cryptology ePrint Archive, Report 2019/354 (2019), https://eprint.iacr.org/2019/354
4. Aumüller, M., Dietzfelbinger, M., Klaue, P.: How good is multi-pivot quicksort? (2016)
5. Beaver, D.: Efficient Multiparty Protocols Using Circuit Randomization. In: CRYPTO '91. Lecture Notes in Computer Science, vol. 576, pp. 420–432. Springer (1992)
6. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: European Symposium on Research in Computer Security. pp. 192–206. Springer (2008)
7. Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic language for privacy-preserving applications. In: Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies. pp. 23–26 (2013)
8. Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis. In: International Conference on Financial Cryptography and Data Security. pp. 57–64. Springer (2012)
9. Boura, C., Chillotti, I., Gama, N., Jetchev, D., Peceny, S., Petric, A.: High-precision privacy-preserving real-valued function evaluation. IACR Cryptol. ePrint Arch. **2017:1234**
10. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. pp. 35–50 (01 2010). https://doi.org/10.1007/978-3-642-14577-3$_6$
11. Chang, C., Melhem, R.: Arbitrary size benes networks. Parallel Processing Letters **07** (05 1997). https://doi.org/10.1142/S0129626497000292
12. Cheng, K., Fan, T., Jin, Y., Liu, Y., Chen, T., Yang, Q.: Secureboost: A lossless federated learning framework. arXiv preprint arXiv:1901.08755 (2019)
13. Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. IACR Cryptol. ePrint Arch. **2019:695**
14. Cho, H., Wu, D.J., Berger, B.: Secure genome-wide association analysis using multiparty computation. Nature biotechnology **36**(6), 547–551 (2018)
15. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Annual Cryptology Conference. pp. 643–662. Springer (2012)
16. Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: 22nd Annual Network and Distributed System Security Symposium, NDSS (2015)

17. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved primitives for MPC over mixed arithmetic-binary circuits. In: 40th Annual International Cryptology Conference, CRYPTO. Lecture Notes in Computer Science, vol. 12171, pp. 823–852 (2020)
18. Feng, Z., Xiong, H., Song, C., Yang, S., Zhao, B., Wang, L., Chen, Z., Yang, S., Liu, L., Huan, J.: Securegbm: Secure multi-party gradient boosting. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 1312–1321. IEEE (2019)
19. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., Wernsing, J.: Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In: International Conference on Machine Learning. pp. 201–210 (2016)
20. Goldschmidt, R.: Applications of division by convergence. MIT Ph.D. Thesis (1964)
21. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: International Conference on Information Security and Cryptology. pp. 202–216. Springer (2012)
22. Harris, D.: A taxonomy of parallel prefix networks. In: The Thirty-Seventh Asilomar Conference on Signals, Systems & Computers, 2003. vol. 2, pp. 2213–2217. IEEE (2003)
23. Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. IACR Cryptol. ePrint Arch. **2011:122**
24. Juvekar, C., Vaikuntanathan, V., Chandrakasan, A.: GAZELLE: A low latency framework for secure neural network inference. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1651–1669 (2018)
25. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1575–1590 (2020)
26. Keller, M., Orsini, E., Scholl, P.: Mascot: faster malicious arithmetic secure computation with oblivious transfer. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 830–842 (2016)
27. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: EUROCRYPT 2018. Lecture Notes in Computer Science, vol. 10822, pp. 158–189 (2018)
28. Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure mpc with dishonest majority. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 549–560 (2013)
29. Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free xor gates and applications. In: International Colloquium on Automata, Languages, and Programming. Lecture Notes in Computer Science, vol. 5126, pp. 486–498. Springer (2008)
30. Liu, J., Juuti, M., Lu, Y., Asokan, N.: Oblivious neural network predictions via minionn transformations. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 619–631 (2017)
31. Makri, E., Rotaru, D., Vercauteren, F., Wagh, S.: Rabbit: Efficient comparison for secure multi-party computation. IACR Cryptol. ePrint Arch. **2021:119**, https://eprint.iacr.org/2021/119
32. Mohassel, P., Rindal, P.: ABY3: A mixed protocol framework for machine learning. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 35–52 (2018)
33. Mohassel, P., Rosulek, M., Trieu, N.: Practical privacy-preserving k-means clustering. Proceedings on Privacy Enhancing Technologies **2020**(4), 414–433 (2020)

34. Mohassel, P., Zhang, Y.: SecureML: A system for scalable privacy-preserving machine learning. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 19–38. IEEE (2017)
35. Patra, A., Schneider, T., Suresh, A., Yalame, H.: ABY2.0: improved mixed-protocol secure two-party computation. IACR Cryptol. ePrint Arch. **2020:1225**, https://eprint.iacr.org/2020/1225
36. Patra, A., Suresh, A.: BLAZE: blazing fast privacy-preserving machine learning. In: 27th Annual Network and Distributed System Security Symposium, NDSS (2020)
37. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
38. Pullonen, P., Siim, S.: Combining secret sharing and garbled circuits for efficient private ieee 754 floating-point computations. In: International Conference on Financial Cryptography and Data Security. pp. 172–183. Springer (2015)
39. Riazi, M.S., Weinert, C., Tkachenko, O., Songhori, E.M., Schneider, T., Koushanfar, F.: Chameleon: A hybrid secure computation framework for machine learning applications. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security. pp. 707–721 (2018)
40. Wagh, S., Gupta, D., Chandran, N.: SecureNN: 3-party secure computation for neural network training. Proceedings on Privacy Enhancing Technologies **2019**(3), 26–49 (2019)
41. Waksman, A.: A permutation network. Journal of the ACM pp. 159–163 (1968)
42. Wikipedia: Hessian matrix. https://en.wikipedia.org/wiki/Hessian_matrix
43. Wikipedia: Newton's method. https://en.wikipedia.org/wiki/Newton%27s_method
44. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 220–250. Springer (2015)

## Supplementary Material

### Seeded triple generation

In order to reduce the communication complexity overall, we use a standard technique based on deterministic pseudorandom number generators ($\mathsf{DRBG}$), that takes as input a uniformly random $\mathsf{seed}$ of size $\ell$ and some parameters $\mathsf{params}$, and outputs a deterministic pseudorandom stream of bytes: $\mathsf{randOut}$ of length $\ell'$, where $\ell' \gg \ell$. Suppose that the trusted dealer needs to generate and communicate shares for a precomputed $N$-by-$k$ matrix. Instead of sending the shares, the dealer send a much smaller seed that is sufficient for the players to reconstruct the shares in the online phase using $\mathsf{DRBG}$.

There are two different cases according to whether we generate and communicate masks associated to a MPC variable (e.g., $\lambda$ and $\mu$ in Beaver multiplication that are independent of previously generated masks) or whether we generate and communicate shares for a precomputed matrix that depends on previously generated random matrices (e.g., $\lambda \cdot \mu$ in Beaver):

1. The precomputed matrix $\lambda$ should be a uniformly random matrix independent of the previously generated precomputed data. To achieve this, the trusted dealer generates $n$ independent random seeds $\mathsf{seed}_i$, one for each player, and sends $\mathsf{seed}_i$ to the $i$th player who computes the corresponding secret share $\lambda_i = \mathsf{DRBG}(\mathsf{seed}_i, \mathsf{params})$ in the online phase. Here, $\lambda = \sum_{i=1}^{n} \lambda_i$ and the seeds are specific to the matrix $\lambda$. The communication overhead here is only the seed $\mathsf{seed}_i$, i.e., $O(\ell)$ bits per player.

2. The random matrix $\nu$ depends on previously precomputed random matrices (e.g., $\nu = \lambda \cdot \mu$ in Beaver multiplication). The space $\mathcal{R}$ of $N$-by-$k$ matrices is then decomposed as a direct sum of $n$ vector subspaces $\mathcal{R}_1, \ldots, \mathcal{R}_n$ of roughly the same dimension. For instance, a $N$-by-$k$ matrix can be split into $n$ row blocks of roughly the same size $(N/n)$-by-$k$. Then $\mathcal{R}_i$ is the subspace of all matrices that have 0's in all positions except for the $i$th row block. We then have a direct sum decomposition $\mathcal{R} = \bigoplus_{i=1}^{n} \mathcal{R}_i$ (here, $\bigoplus$ means direct sum of vector spaces and not XOR). For each player, the trusted dealer generates a seed $\mathsf{seed}_i$ and a matrix $\gamma_i = \mathsf{DRBG}'(\mathsf{seed}_i, \mathsf{params}, i) \in \bigoplus_{j \neq i} \mathcal{R}_j$. Here, $\mathsf{DRBG}'(\mathsf{seed}_i, \mathsf{params}, i)$ is a pseudorandom function that uses $\mathsf{DRBG}$ and that outputs a $N$-by-$k$ matrix whose projection to the $i$th direct summand subspace $\mathcal{R}_i$ is zero. The trusted dealer then computes $c = \nu - \sum_{i=1}^{n} \gamma_i$, decomposes $c = c_1 + \cdots + c_n$ according to $\bigoplus \mathcal{R}_i$ (i.e., $c_i$ is the projection of $c$ to the subspace $\mathcal{R}_i$) and sends $(\mathsf{seed}_i, c_i)$ to the $i$th player. Locally, the $i$th player reconstructs its own share $\nu_i = \mathsf{DRBG}'(\mathsf{seed}_i, \mathsf{params}, i) + c_i = \gamma_i + c_i$, thus giving the plaintext mask $\nu = \sum_{i=1}^{n} \nu_i = \sum_{i=1}^{n} \gamma_i + c$. The communication overhead here is $O(\ell + \ell'/n)$ bits per player.

This design thus reduces the communication overhead at least by a factor $n$. For the DRBG, we rely on the AES-CTR DRBG implementation of `Mbed TLS`[10], taking advantage of the native AES instruction set of the processor.

### Linear combination and Beaver multiplication

*Linear combination.* The goal of the linear combination is to compute a secret shares of $z = \sum_{i=1}^{m} c^{(i)} \cdot x^{(i)}$, where the coefficients $c^{(1)}, \ldots, c^{(m)}$ are public real numbers given as double-precision floating point numbers and $[\![x^{(1)}]\!], \ldots, [\![x^{(m)}]\!]$ secret shares with corresponding parameters $M^{(i)}_{\text{msb}}$, $p^{(i)}_{\text{msb}}$, $p^{(i)}_{\text{lsb}}$.

Let $\beta^{(i)} := \lfloor 2^{58} \cdot c^{(i)} \rceil$ [11] and let

$$\alpha^{(i)} := 2^{-v_2\left(\beta^{(i)}\right)}\beta^{(i)} \quad \text{and} \quad \ell^{(i)} := 58 - v_2\left(\beta^{(i)}\right),$$

where for a prime $p$, $v_p(d)$ denotes $p$-adic valuation of an integer $d \in \mathbb{Z}$. In practice, these integers are represented as 64-bit signed integers. The public vector $\left(\frac{\alpha^{(1)}}{2^{\ell^{(1)}}}, \ldots, \frac{\alpha^{(m)}}{2^{\ell^{(m)}}}\right) \neq 0$ will be our approximation of the coefficients.

The compiler must provide two working parameters: $M^{w}_{\text{msb}}$ and $p^{w}_{\text{lsb}}$. These are selected in such a way that they satisfy the following properties: 1.) The lift to the `ModReal` shares class $\mathcal{M}_{M^{w}_{\text{msb}}, p^{w}_{\text{lsb}}}$ of the inputs ensures that the individual scalar products $\frac{\alpha^{(i)}}{2^{\ell^{(i)}}} \cdot x^{(i)}$ can be computed without overflow and keeping sufficient precision; 2.) The sum of all secret shares (there are $\ell$ of them) of all individual terms (scalar products; there are $m$ of them) can be computed in this class without an overflow.

The idea of the algorithm is: For each $1 \leq i \leq m$, the $i$th input variable is first lifted to a `ModReal` number with parameters $M^{w}_{\text{msb}} + \ell^{(i)}$ and $p^{w}_{\text{lsb}} + \ell^{(i)}$. When $M^{w}_{\text{msb}} + \ell^{(i)} > M^{(i)}_{\text{msb}}$, this requires a lift triple (involving the variable's mask $\lambda$, and two ephemeral precomputed data $b$ and $\nu$). Then, the multiplication by $\alpha^{(i)}/2^{\ell^{(i)}}$ yields a value of parameters $M^{w}_{\text{msb}}$ and $p^{w}_{\text{lsb}}$, which gets accumulated with the other values. Finally, the sum is casted to the output parameters.

*Beaver Multiplications.* For the multiplication of two secret shares $[\![x]\!]_{M^{x}_{\text{msb}}, p^{x}_{\text{lsb}}}$ and $[\![y]\!]_{M^{y}_{\text{msb}}, p^{y}_{\text{lsb}}}$ we use the Beaver's method [5]. The goal is to compute $[\![z]\!]_{M^{z}_{\text{msb}}, p^{z}_{\text{lsb}}}$ with corresponding output parameters $M^{z}_{\text{msb}}$ and $p^{z}_{\text{lsb}}$. These parameters are typically determined by the compiler during the static analysis. The compiler also needs to provide working parameters for the intermediate variables:

- $(M^{AW}_{\text{msb}}, p^{AW}_{\text{lsb}})$ - lifting parameters for the masked value $a = x + \lambda$,

[10] https://tls.mbed.org

[11] The choice of 58 above in order to resemble the plaintext of type `float64`, whereby the mantissa is 53 bits

– $(M^{BW}_{\text{msb}}, p^{BW}_{\text{lsb}})$ - lifting parameters for the masked value $b = y + \mu$

The working parameters for $x$ and $y$ are generated in a such a way that they satisfy the following properties: 1.) the multiplication can be computed in $\mathcal{M}_{M^W_{\text{msb}}, p^W_{\text{lsb}}}$ without an overflow; 2.) $x$ and $y$ are lifted to numerical window of the same size; 3.) preserve the minimal precision between the two inputs and the output. The idea of the algorithm is: First the players mask $x$ and $y$ and reveal the masked values $a = x + \lambda$ and $b = y + \mu$. After that, $a$ and $b$ are lifted to the corresponding working parameters $(M^{AW}_{\text{msb}}, p^{AW}_{\text{lsb}})$ and $(M^{BW}_{\text{msb}}, p^{BW}_{\text{lsb}})$. The output is computed using Beaver method: $[\![z]\!] = a \cdot b - a \cdot [\![\mu]\!] - [\![\lambda]\!] \cdot b + [\![\lambda\mu]\!] = [\![x]\!] \cdot b - a \cdot [\![\mu]\!] + [\![\lambda\mu]\!]$ (note that the 3 terms representation is equivalent and 33% more efficient) and at the end we lift to the output parameters $(M^z_{\text{msb}}, p^z_{\text{lsb}})$.

### Logreg benchmarks

Tables 6, 7 and 8 provide computation overhead for datasets with different number of rows and features for 2, 3 and 5 players respectively. The counter-intuitive decrease in triples size with the increase of number of players is due to seeded triples generation.

| | | RAM | | Communication | | CPU | | End-to-end | |
|---|---|---|---|---|---|---|---|---|---|
| Features count | Input size $\times 10^6$ | Offline | Online | Triples | Network | Offline | Online | Offline | Online |
| | | MB | | | | seconds | | | |
| 10 | 0.01 | 55 | 51 | 103 | 79 | 5.4 | 2.4 | 5.4 | 2.6 |
| 50 | | 81 | 142 | 292 | 344 | 9.5 | 6.6 | 9.5 | 8.2 |
| 100 | | 140 | 257 | 533 | 682 | 16.2 | 15.1 | 16.2 | 17.8 |
| 10 | 0.1 | 133 | 230 | 1025 | 784 | 63.6 | 21.8 | 63.6 | 23.9 |
| 50 | | 491 | 784 | 2890 | 3412 | 106.7 | 64.8 | 106.7 | 80.3 |
| 100 | | 939 | 1449 | 5228 | 6702 | 171.6 | 149.8 | 171.6 | 175.7 |
| 10 | 0.5 | 566 | 860 | 5120 | 3919 | 159.9 | 101.3 | 159.9 | 113.7 |
| 50 | | 2359 | 3551 | 14431 | 17044 | 377.9 | 327.1 | 377.9 | 403.7 |
| 100 | | 4596 | 6897 | 26077 | 33457 | 724.9 | 765.1 | 724.9 | 890.9 |
| 10 | 1 | 1123 | 1673 | 10238 | 7837 | 271.9 | 188.7 | 271.9 | 216.3 |
| 50 | | 4699 | 7184 | 28858 | 34085 | 711.8 | 644.9 | 711.8 | 799.9 |
| 100 | | 9170 | 13806 | 52139 | 66901 | 1425.5 | 1550.5 | 1425.5 | 1797.9 |

**Table 6.** Logistic regression execution times and communication sizes for 2 players and different dataset sizes.

| | | RAM | | Communication | | CPU | | End-to-end | |
|---|---|---|---|---|---|---|---|---|---|
| Features count | Input size $\times 10^6$ | Offline | Online | Triples | Network | Offline | Online | Offline | Online |
| | | MB | | | | seconds | | | |
| 10 | 0.01 | 55 | 55 | 69 | 157 | 5.9 | 2.8 | 5.9 | 3.5 |
| 50 | | 81 | 166 | 195 | 688 | 11.2 | 8.6 | 11.2 | 12.6 |
| 100 | | 140 | 292 | 356 | 1364 | 19.4 | 19.7 | 19.4 | 27.1 |
| 10 | 0.1 | 133 | 264 | 684 | 1568 | 66.0 | 26.4 | 66.0 | 34.2 |
| 50 | | 528 | 826 | 1927 | 6823 | 120.9 | 84.1 | 120.9 | 124.1 |
| 100 | | 937 | 1779 | 3486 | 13404 | 206.2 | 193.6 | 206.2 | 266.6 |
| 10 | 0.5 | 601 | 949 | 3413 | 7837 | 184.3 | 119.6 | 184.3 | 161.0 |
| 50 | | 2360 | 4094 | 9621 | 34089 | 486.4 | 435.7 | 486.4 | 632.6 |
| 100 | | 4596 | 8041 | 17386 | 66914 | 932.3 | 1028.8 | 932.3 | 1380.7 |
| 10 | 1 | 1108 | 1804 | 6826 | 15673 | 332.1 | 236.2 | 332.1 | 319.6 |
| 50 | | 4699 | 8100 | 19239 | 68170 | 929.1 | 870.1 | 929.1 | 1264.2 |
| 100 | | 9170 | 15422 | 34760 | 133802 | 1829.7 | 2077.7 | 1829.7 | 2777.2 |

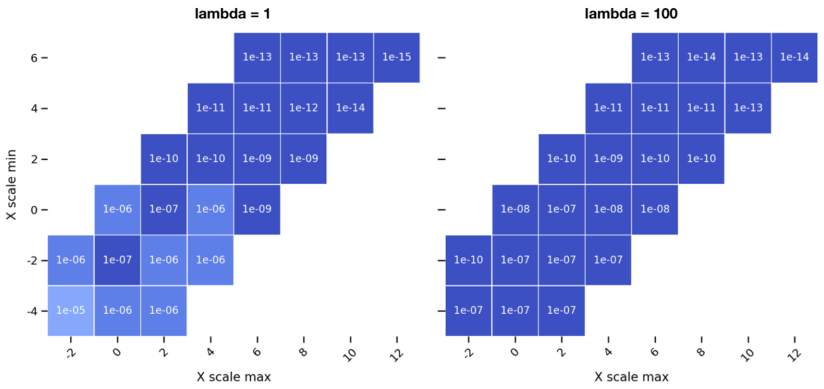
**Table 7.** Logistic regression execution times and communication sizes for 3 players and different dataset sizes.

| | | RAM | | Communication | | CPU | | End-to-end | |
|---|---|---|---|---|---|---|---|---|---|
| Features count | Input size $\times 10^6$ | Offline | Online | Triples | Network | Offline | Online | Offline | Online |
| | | MB | | | | seconds | | | |
| 10 | 0.01 | 55 | 62 | 41 | 314 | 7.2 | 3.3 | 7.2 | 5.2 |
| 50 | | 81 | 197 | 117 | 1377 | 15.3 | 11.0 | 15.3 | 20.3 |
| 100 | | 140 | 373 | 214 | 2728 | 27.5 | 26.0 | 27.5 | 43.5 |
| 10 | 0.1 | 133 | 289 | 411 | 3136 | 80.0 | 31.6 | 80.0 | 51.4 |
| 50 | | 490 | 1202 | 1157 | 13647 | 163.8 | 114.0 | 163.8 | 204.9 |
| 100 | | 937 | 2155 | 2093 | 26808 | 284.4 | 264.9 | 284.4 | 435.3 |
| 10 | 0.5 | 565 | 1189 | 2048 | 15674 | 255.1 | 154.1 | 255.1 | 253.9 |
| 50 | | 2361 | 5261 | 5773 | 68177 | 717.2 | 601.9 | 717.2 | 1049.7 |
| 100 | | 4596 | 9232 | 10432 | 133829 | 1368.0 | 1424.8 | 1368.0 | 2255.1 |
| 10 | 1 | 1108 | 2346 | 4096 | 31347 | 467.9 | 301.1 | 467.9 | 502.1 |
| 50 | | 4699 | 9809 | 11544 | 136340 | 1394.2 | 1213.1 | 1394.2 | 2106.7 |
| 100 | | 9170 | 18659 | 20857 | 267605 | 2689.8 | 2871.7 | 2689.8 | 4527.4 |

**Table 8.** Logistic regression execution times and communication sizes for 5 players and different dataset sizes.

**Numerical precision of $\boldsymbol{\theta_{mpc}}$ against $\boldsymbol{\theta_{skl}}$.** We also assessed the *numerical precision* of $\theta_{mpc}$ against $\theta_{skl}$ by computing the maximum absolute difference of the $k$ coefficients of theta, $\delta_\theta = max(|\theta_{mpc} - \theta_{skl}|)$. Fig.4 shows the highest discrepancy $\delta_\theta$ in the model coefficients between MPC and plaintext, for datasets yielding approximately the same log loss, $\mathcal{L}_{diff} \approx 0$. We again take the worst result over 5 runs and apply truncation formatting for readability. The obtained precision depends on the scale of $\theta_{mpc}$, the higher the scale the lower the preci-

sion, which is unavoidable as all the model coefficients share the same fix-point. Note however that losses of precision on smaller coefficients has negligible or no impact on the predictions $\hat{y}$, as they contribute the least in the calculation. For a dataset $X$ with column scales $s_j \in [0..6]$ and $\lambda = 1$, we get a $\delta_\theta$ of $10^{-9}$, thus matching the plaintext model at least 8 digits after the decimal point.




h

**Fig. 4.** Theta max absolute difference ($\delta_\theta$) Manticore vs scikit-learn

**MP-SPDZ vs `Manticore` .** In a second experiment, we have compared the logistic regression implementations of `Manticore` and MP-SPDZ [25]. Besides execution metrics, we have compared the precision of these MPC implementations to plaintext learning via `scikit-learn` [37]. We have generated a synthetic dataset $X$ with 30000 rows and 10 features together with a vector $y$ of labels. The values of the matrix $X$ are uniformly random in the interval $[-4, 4]$ and $y$ consists of uniformly random binary values. Two more datasets with non-normalized and correlated features were generated from $X$: in the first one, $X'$, we have re-scaled a column of $X$ by $2^8$ and in the second one, $X|X$, we have horizontally stacked 2 times matrix $X$.

The Manticore logistic-regression is configured with 10 IRLS followed by 2 classical iterations. The MP-SPDZ implements a mini-batch stochastic-gradient descent (SGD). We use the sample logistic-regression code from MP-SPDZ github[12] (commit `15d179a`). We have employed 4 configurations:

- `mp-spdz a` and `mp-spdz b` use a 5 piece-wise sigmoid approximation with 10 and respectively 100 iterations,
- `mp-spdz c` and `mp-spdz d` use exact sigmoid with 10 and respectively 100 iterations.

[12] https://github.com/data61/MP-SPDZ

These configurations use the same batch size 128. For other parameters, we have tried our best to obtain similar configurations between `Manticore` and MP-SPDZ implementations (no regularization, no normalization and no intercept is used). We have compiled the logistic-regression with arithmetic modulo $2^k$ and extended daBits (`-R 64 -Y` compile arguments). Replicated secret-sharing execution environment with 3 parties (honest-majority) is used (the `ring.sh` execution script).

| Dataset | Method | Log-loss | Exec. time | Comm. |
|---|---|---|---|---|
| $X$ | `manticore` | **0.445** | 12.4 | 512 |
| | `mp-spdz a` | 0.449 | 5.5 | 263 |
| | `mp-spdz b` | 0.449 | 35.4 | 1020 |
| | `mp-spdz c` | **0.445** | 43.4 | 2697 |
| | `mp-spdz d` | **0.445** | 414.9 | 25352 |
| $X'$ | `manticore` | **0.445** | 12.5 | 512 |
| | `mp-spdz a` | 8.549 | 5.5 | 263 |
| | `mp-spdz b` | 4.689 | 35.6 | 1020 |
| | `mp-spdz c` | 17.102 | 43.5 | 2697 |
| | `mp-spdz d` | 3.821 | 415.8 | 25352 |
| $X\|X$ | `manticore` | **0.445** | 12.8 | 539 |
| | `mp-spdz a` | 0.652 | 5.5 | 264 |
| | `mp-spdz b` | 0.695 | 36.3 | 1033 |
| | `mp-spdz c` | 0.825 | 43.6 | 2698 |
| | `mp-spdz d` | 0.819 | 415.2 | 25365 |

**Table 9.** MP-SPDZ vs Manticore precision, execution time (seconds) and communication overhead (MB) comparison. Logistic-loss values which differ by less than $10^{-3}$ from the `scikit-learn` are highlighted in bold font.

In table 9 are given execution metrics for different configurations of tested logistic-regression implementations. Model quality is measured by the logistic-regression objective function, i.e. the logistic loss ("logloss" column). The execution time given in column "Exec. time" corresponds to offline and online phases. The network size is given in column "Comm.". `Manticore` and MP-SPDZ model qualities for dataset $X$ are similar to plaintext model (log-loss difference under $10^{-2}$). The `Manticore` implementation is precise (same log-loss as the plaintext model) when the dataset is not normalized ($X'$) or has correlated features ($X|X$) as opposed to MP-SPDZ. We suppose that the mini-batch SGD algorithm implemented in MP-SPDZ is not well suited for such type of datasets. MP-SPDZ is faster and has lower communication when approximate sigmoid and 10 SGD iterations are used, although for same model precision the `Manticore` is always better.