EFFICIENT BUILDING BLOCKS FOR SECURE MULTIPARTY COMPUTATION AND THEIR APPLICATIONS

by

Donghang Lu

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Aniket Kate, Chair

School of Computer Science

Dr. Mikhail Atallah

School of Computer Science

Dr. Hemanta K. Maji

School of Computer Science

Dr. Yexiang Xue

School of Computer Science

Approved by:

Dr. Kihong Park

I dedicate this dissertation to my loving parents, who provide significant support to make everything possible, and to my friends, who helped me go through all the ups and downs.

ACKNOWLEDGMENTS

It is my great honor to express my gratitude to a large number of people who contribute to my Ph.D. journey.

My most sincere gratitude goes to my advisor, Professor Aniket Kate, for his support and guidance all the time. It means everything to me to become a better researcher, and to become a better person. The discussion with him always comes with creative ideas and enlightenment, and many works in this Dissertation are impossible without him.

I also want to express my thanks to my parents, they are the strongest support during my whole Ph.D. journey, actually my whole life, and provide the most significant emotional support.

Heartful thanks to my committee members Professor Aniket Kate, Professor Hemanta K. Maji, Professor Mikhail Atallah, and Professor Yexiang Xue for their valuable advice and guidance.

I'm grateful to have spent time with all my lab-mates and friends, they give me endless happiness and motivation to move forward. I would like to thank Jiayi Meng, Pedro Moreno Sanchez, Duc Le, Mohsen Minaei, Easwar Mangipudi, Debajyoti Das, Adithya Bhat, Tiantian Gong, Albert Yu, Zhongtang Luo, Mohammad Hassan Ameri, Sihao Yin, Shivam Bajpayi, Zifu Wei, Xinghai Hong, Ke Quan, Cheng Zhang, Sikai Chen, Yue Leng.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Secure multi-party computation (MPC) enables mutually distrusting parties to compute securely over their private data. It is a natural approach for building distributed applications with strong privacy guarantees, and it has been used in more and more real-world privacy-preserving solutions such as privacy-preserving machine learning, secure financial analysis, and secure auctions.

The typical method of MPC is to represent the function with arithmetic circuits or binary circuits, then MPC can be applied to compute each gate privately. The practicality of secure multi-party computation (MPC) has been extensively analyzed and improved over the past decade, however, we are hitting the limits of efficiency with the traditional approaches as the circuits become more complicated. Therefore, we follow the design principle of identifying and constructing fast and provably-secure MPC protocols to evaluate useful high-level algebraic abstractions; thus, improving the efficiency of all applications relying on them.

To begin with, we construct an MPC protocol to efficiently evaluate the powers of a secret value. Then we use it as a building block to form a secure mixing protocol, which can be directly used for anonymous broadcast communication. We propose two different protocols to achieve secure mixing offering different tradeoffs between local computation and communication. Meanwhile, we study the necessity of robustness and fairness in many use cases, and provide these properties to general MPC protocols. As a follow-up work in this direction, we design more efficient MPC protocols for anonymous communication through the use of permutation matrices. We provide three variants targeting different MPC frameworks and input volumes. Besides, as the core of our protocols is a secure random permutation, our protocol is of independent interest to more applications such as secure sorting and secure two-way communication.

Meanwhile, we propose the solution and analysis for another useful arithmetic operation: secure multi-variable high-degree polynomial evaluation over both scalar and matrices. Secure polynomial evaluation is a basic operation in many applications including (but not limited to) privacy-preserving machine learning, secure Markov process evaluation, and non-linear function approximation. In this work, we illustrate how our protocol can be used to efficiently

evaluate decision tree models, with both the client input and the tree models being private. We implement the prototypes of this idea and the benchmark shows that the polynomial evaluation becomes significantly faster and this makes the secure comparison the only bottleneck. Therefore, as a follow-up work, we design novel protocols to evaluate secure comparison efficiently with the help of pre-computed function tables. We implement and test this idea using Falcon, a state-of-the-art privacy-preserving machine learning framework and the benchmark results illustrate that we get significant performance improvement by simply replacing their secure comparison protocol with ours.

1. INTRODUCTION

Secure multi-party computation (MPC) allows mutually distrusting parties to securely compute some functions, with their private input remaining hidden from other parties. Informally, in a system of $n > 1$ mutually distrusting parties, an MPC protocol allows them to "securely" evaluate any agreed-on function f of their private inputs, in the presence of a centralized adversary controlling at most any $t < n$ parties[1]. In recent years, MPC has made great strides toward practical implementation and real-world deployment. Consequently, it becomes a natural approach to build distributed applications with strong privacy guarantees. MPC has been applied to various applications such as privacy-preserving machine learning (PPML), online auctions, and online elections.

Theoretically, any computation/function can be represented as a Boolean or arithmetic circuit, and a secure MPC protocol can proceed by inductively composing the secure protocols for the atomic elementary Boolean/arithmetic gates. Although this approach is complete, it introduces significant overheads. Therefore, to make MPC more practical, we design MPC protocols for high-level building blocks, instead of only elementary gates.

In this thesis, we follow two directions to make MPC more efficient and practical: (1) Design fast and provably-secure MPC protocols to evaluate useful high-level algebraic abstractions. (2) Build up applications that get significant performance gain by using the MPC protocols we design. We focus on two prominent applications: anonymous broadcast and privacy-preserving machine learning. Multiple highly-efficient MPC building blocks are designed to improve the performance of these two applications. What's more, the building blocks we build can also be extended and used in more applications.

1.1 Our Contribution

1.1.1 Secure Mixing Protocols

Millions of users employ the Tor [1] network to protect the anonymity of their communication over the Internet today. However, Tor can only provide a weak form of anonymity against traffic analysis [2] and has been successfully attacked using strong adversaries [3], [4].

[1] ↑For different system adversary settings, MPC may have stricter requirements such as $t < \frac{n}{2}$ or $t < \frac{n}{3}$.

In this thesis, we make use of MPC to achieve anonymous communication through a secure mixing protocol. We design and implement secure mixing protocols in the setting of MPC system-as-a-service (MPSaaS), where a network of servers continuously processes encrypted inputs submitted by clients.

In [5], we propose two protocols for the purpose of mixing. The first protocol is *switching network*, which has $O(log^2(k))$ round complexity for mixing k client input with cheap local computation. The second protocol is *Powermixing*, which has constant round complexity but with high local computation. These two protocols provide different trade-offs and their performance varies in different server settings and network settings.

The core part of the Powering protocol is a high-level arithmetic operation: Given secret shared value $[x]$, compute all its powers $[x^2], [x^3], \ldots, [x^k]$. This building block may be of independent interest and can be used in other applications.

When building up mixing protocols, we also focus on robustness and fairness. In the context of anonymous communication, unfair MPC could be catastrophic since an adversary could link the messages of clients who retry to send their message in a new or restarted instance. Thus the primary goal of our work is to fill this gap by advancing robustness in practical MPC implementations and demonstrating the result through a novel robust message mixing service.

1.1.2 Secure Random Permutation

The evaluation results illustrate that Powermixing achieves fabulous performance when dealing with a small number of inputs. However, when the number of inputs increases, the performance drops fast due to the $O(k^3)$ online computation complexity. Therefore, we conduct further researches to design mixing protocols that are more scalable for a large number of inputs. As a follow-up work of [5], we design RPM, a new MPC primitive to achieve secure random permutation. Anonymous broadcast can be achieved by applying an unknown random permutation to the input messages. The core idea is that the permutation can be represented using permutation matrices. Therefore, as long as servers prepare the

permutation matrices properly in the offline phase, the online phase protocol can be simply some inner products.

We provide three variants of our protocols, such that different variants suit different MPC frameworks and input volumes. Our protocols can be implemented on almost all existing MPC frameworks due to the modular design. For MPC frameworks with efficient inner product protocol implemented, the first variant of our protocols can provide a blazing-fast online phase. If the underlying MPC framework does not have an efficient inner product protocol, the second variant becomes the better choice. When dealing with a large volume of inputs, the first two variants suffer from a slow offline phase, therefore we design the third variant which can handle large input volume efficiently in both the online phase and the offline phase.

1.1.3 Secure Polynomial Evaluation

We design 2-round protocols for evaluating multivariate polynomials [6], and the communication complexity (i.e., the number of communicated bits) only increases linearly with the number of variables and is independent of polynomial degree. Our highly parallelizable protocols employ specially crafted pre-computations and offer significant improvements over the state-of-the-art techniques that require communication rounds logarithmic in the number of variables. We also present 2-round protocols for evaluating univariate polynomials, which extends a secure exponentiation protocol by Damgård et al. [7].

To illustrate how our protocols can be used to solve real-world problems, We present a solution for privacy-preserving decision tree evaluation as an application, and it shows that we can achieve high-depth decision tree evaluation within a reasonable time. To the best of our knowledge, our solution is the first to support general n-party setting and high-depth tree evaluations simultaneously.

We implement our protocols using the HoneyBadgerMPC library [5] for robust execution in the asynchronous setting. For a depth 8 complete decision tree model, we can evaluate it with 12 seconds under 4-party setting or 13 seconds under 7-party setting.

1.1.4 Secure Comparison

In [6], we find out that the secure comparison protocol becomes the bottleneck of our secure decision tree evaluation. What's more, an efficient secure comparison solution has the potential for significant real-world impact through its broad applications. Therefore, we present a secure comparison protocol with blazing-fast online phase performance by leveraging function tables in a creative way. The general idea is that we construct the function table of secure comparison functionality in the offline phase, such that the online phase is as simple as a table lookup. We design efficient map-reduce solutions such that the size of the function table is reasonable and practical.

We choose secure neural network training and inference as the high-level application and implement our secure comparison protocol on the Falcon [8] framework. The benchmark shows that our secure comparison protocol is 2× faster than Falcon's secure comparison with 4× cheaper communication. Therefore, we obtain around 1.3× performance improvement on neural network training/inference by just replacing their secure comparison protocol with ours.

1.1.5 Secure Matrix Operation

We present a 4-round protocol in [6] for secure matrix powering/exponentiation and make use of it to evaluate univariate polynomials over matrices securely. Our protocol is not only highly parallelizable but also with a communication complexity independent of the exponent. For multivariate polynomials over matrices, we evaluate two alternatives for securely multiplying an arbitrary number of matrices. Using these protocols, we can securely evaluate the terms of a matrix polynomial in parallel offering a trade-off in terms of communication complexity and round complexity.

Despite the conceptual similarity, our protocols for secure polynomial evaluation for scalars over finite fields and those over matrices are inherently different as the multiplication of finite field elements is commutative, while matrix multiplication is not commutative in general.

We use our matrix powering protocol to solve the evaluation of a Markov process. Furthermore, we show how one can use this computation to perform credit risk analysis in financial domains. The benchmark shows that in the 4-party setting, we can evaluate the power of 10×10 transition matrices (with the exponent being 1024) in (roughly) half a second.

1.2 Thesis Structure

The rest of the thesis is organized as follows: In Chapter 2, we introduce our first research work for anonymous communication: Asynchromix. Then in Chapter 3, we keep the same direction and present the follow-up work RPM. In Chapter 4, we talk about our research works in another research direction: Polymath, which provides efficient MPC polynomial evaluation protocols. Then we talk about our follow-up work to speed up the secure comparison protocols in Chapter 5. Finally, in Chapter 6, we conclude the research works so far and describes some future works and directions.

2. ASYNCHROMIX: ANONYMOUS COMMUNICATION WITH ROBUSTNESS

2.1 The Motivation and the Background

2.1.1 MPC Based on Shamir Secret Sharing

Our standard MPC setting involves n parties $\{P_1, \ldots, P_n\}$, where up to $t < n/3$ of those can be compromised by a Byzantine adversary. HoneyBadgerMPC relies on many standard components for MPC [9]–[12] based on Shamir secret-sharing [13]. Here, we detail the most relevant techniques and notation.

Shamir Secret Sharing and Reconstruction

Notation

For prime p and a secret $s \in \mathbb{F}_p$, $[s]_t$ denotes Shamir secret sharing (SSS) with threshold t (i.e., a t-sharing). Specifically, a degree-t polynomial $\phi : \mathbb{F}_p \to \mathbb{F}_p$ is sampled such that $\phi(0) = s$. The share $[s]_t^{(\mathrm{i})}$ is the evaluation $\phi(\mathrm{i})$. The superscript and/or subscript of a share may be omitted when clear from the context.

Robust interpolation of polynomials

Reconstructing a secret s from $[s]$ (denoted as $\mathsf{Open}()$) requires interpolating the polynomial ϕ from shares received from other parties. Since we want to achieve security against an active (Byzantine) attacker, up to t of the shares may be erroneous. Furthermore, in an asynchronous network, we cannot distinguish a crash fault from an intentional withholding of data and can consequently only expect to receive shares from $n - t$ parties in the worst case.

Figure 2.1 outlines the standard approach [9], [11], [12], [14] for robust decoding in this setting, Robust-Interpolate. First, we optimistically attempt to interpolate a degree-t polynomial ϕ after receiving any $t + 1$ shares. If the resulting ϕ coincides with the first $2t + 1$ shares received, then we know it is correct. If the optimistic case fails, we wait to receive more shares to attempt to correct errors. In the worst case, we receive t incorrect shares

and need to wait for $3t + 1$ total shares before we can correct t errors and find a degree-t polynomial that coincides with all $2t + 1$ honest shares. We refer readers to [15] for more details about RSDecode.

Algorithm Robust-Interpolate

- Input: $y_0, ..., y_{n-1}$ symbols, up to t erasures ($y_i \in \mathbb{F}_p \cup \{\perp\}$)
- Output: $a_0, ..., a_t$, coefficients of a degree-t polynomial ϕ, such that $y_i = \phi(\alpha_i)$ for $i \in I$ where $I \subset [1..n]$ and $\mathtt{I} = 2t + 1$, or else $\perp$
- Procedure (case of t erasures):
  1. Interpolate a polynomial ϕ from any $t + 1$ points (y_i, α_i)
  2. Output ϕ if it coincides with all $2t + 1$ points, otherwise output $\perp$
- Procedure (case of $t - e$ erasures):
  1. Run RSDecode decoding to correct up to e errors

Figure 2.1. Robust Polynomial Interpolation

2.1.2 SSS-Based MPC

Linear combinations of SSS-shared secrets can be computed locally, preserving the degree of secret sharing without any necessary interaction between parties. However, in order to be able to realize an arbitrary arithmetic circuit using MPC, we need a way to multiply secrets together. In this work, we use Beaver's trick to multiply two t-sharings $[x]_t$ and $[y]_t$ by consuming a preprocessed Beaver triple. Beaver triples are correlated t-sharings of the form $[a]_t, [b]_t, [ab]_t$, for random $a, b \in \mathbb{F}_p$ which can be used to find $[xy]_t$ by using the following identity:

$$[ab]_t = (a - x)(b - y) + (a - x)[y]_t + (b - y)[x]_t + [xy]_t.$$

If a and b are random and independent of x and y, then Open($[a - x]$) and Open($[b - y]$) do not reveal any information about x or y. Each multiplication then requires the public opening of $(a - x)$ and $(b - y)$ and the spending of a Beaver triple.

We follow the standard online/offline MPC paradigm, where the online phase assumes it can make use of a buffer of preprocessed values that were created during the offline phase. By utilizing precomputed triples and opening $(a - x)$ and $(b - y)$ for many multiplication gates at once, we can process many gates at the same circuit depth simultaneously.

2.1.3 Anonymous Communication from MPC

Secure multi-party computation (MPC) is a natural approach for building distributed applications with strong privacy guarantees. MPC has recently made great strides towards practical implementation and real-world deployment and consequently, several general-purpose compilers (or front-ends [16]) and implementations are now available supporting a range of performance and security tradeoffs [17]–[24]. Recent implementation efforts [23]–[25] have bolstered their security guarantees by focusing on the malicious rather than semi-honest setting (i.e., they tolerate Byzantine faults), and can scale to larger networks (e.g., more than 100 servers) while tolerating an appreciable number of faults. Further, in contrast to early MPC realizations centered around one-off ceremonies [26], [27], there has been increased interest in the MPC system-as-a-service (MPSaaS) [23], [28]–[30] setting, where a network of servers continuously processes encrypted inputs submitted by clients. As scalable and maliciously secure MPSaaS becomes increasingly practical, there's an increasingly more convincing argument that it can be successfully used for highly desirable internet services such as anonymous communication.

AsynchroMix is an application of the MPC-System-as-a-Service (MPSaaS) [23] approach to the problem of anonymous broadcast communication. We consider a typical client-server setting for anonymous communication networks [1], [31], [32], where clients send their confidential messages to server nodes and server nodes mix clients' messages before making them public. We model an asynchronous communication network such that we must not make use of timeouts and do not rely on time-bound parameters to be correctly configured. The communication network is assumed to be under the adversary's control such that the adversary may arbitrarily delay messages, duplicate them, or deliver them out of order. For system liveness, we assume that the adversary cannot drop messages between two honest

parties. The goals of AsynchroMix include Safety (anonymity properties) as well as liveness - the system continues to work. The strong threat model includes a fraction being maliciously corrupted and does not rely on timing assumptions.

2.1.4 The Need for Robustness

In practice, distributed computing protocols should successfully protect against not just benign failures like system churn, but also network partitions and denial of service attacks. Distributed consensus protocols and systems employed in practice (e.g., [33]–[35]) put significant emphasis on achieving this robustness property, and the same also holds for prominent blockchain systems [36], [37]. Various notions of robustness have also been explored in the context of MPC, although we observe that the practical MPC tool-kits [17], [19], [23], [38] available today have not made a similar effort to incorporate this robustness.

In this section, we evaluate the robustness of existing MPC implementations and protocols (summarized in Table 2.1), and use this evaluation to inform the design of HoneyBadgerMPC and AsynchroMix. We focus mainly on three forms of robustness: *fairness*, *guaranteed output*, and *safety in asynchronous communication setting*. In our work, we focus on the MPC-System-as-a-Service model [23], [28]–[30], where clients submit secret inputs to servers for processing. However, in the usual MPC setting, the servers themselves are the clients. Thus for the sake of comparison, in this section, we assume $n = k$ (where n is the number of servers and k is the number of clients). In this evaluation, we leave implicit the need to agree on which inputs to include. In a synchronous network, MPC typically ensures that every honest party's inputs are included [39], while in an asynchronous network it is inherent that up to t honest parties may be left out [11]; to accommodate asynchronous protocols we assume the weaker definition. We also elide discussion of protocols and implementations that offer only semi-honest security, such as PICCO [40] or Fairplay [41], or that rely on trusted hardware [42].

Table 2.1. Summary of Robustness in Active Secure MPC Protocols and Toolkits

Protocol Designs	$t <$	Fairness	Guaranteed Output Online	Guaranteed Output Offline	Asynchronous Safe	Asynchronous Live	Complexity Assumption	Communication Overhead
BGW [39], [43]	$n/3$	✓	✓		✗	✗		quadratic
HN06 [44]	$n/2$	✓	✓		✗	✗	SHE	linear
BH08 [9],DN07 [10]	$n/3$	✓	✓	✓	✗	✗		linear
DN07 [10]	$n/2$	✓	✓	✓	✗	✗	Dlog	linear
DIK+08 [45], [46][1]	$n/8$	✓	✓	✓	✗	✗		linear
COPS15 [47]	$n/2$	✓	✓	✓	✓	✗	HE	quadratic
CHP13[11],CP17[12]	$n/4$	✓	✓	✓	✓	✓		linear
CP15 [14]	$n/3$	✓	✓	✓	✓	✓	SHE	linear
MPC Toolkits								
Viff [38]	$n/3$	✗	✗	✗	✓	✗		quadratic
SPDZ [17], [18], [20]	n	✗	✗	✗	✓	✗	SHE or OT	linear
EMP [21]	n	✗	✗	✗	✓	✗	OT	quadratic
SCALE-MAMBA [19]	$n/2$	✗	✗	✗	✓	✗		quadratic
HyperMPC [23]	$n/3$	✗	✗	✗	✓	✗		linear
CGH+18 [24]	$n/2$	✗	✗	✗	✓	✗		linear
This work								
HoneybadgerMPC [15]	$n/3$	✓	✓	✗	✓	✓		linear

2.2 System Setting

Assume a set of clients $\mathcal{C} = \{C_j\}_{j=1...k_{pop}}$ with input messages m_j, who communicate to a set of n servers, $\{P_i\}_{i=1...n}$. All servers are connected to each other over asynchronous channels, and every client is connected to all servers over asynchronous channels. The messages themselves are fixed sizes of $\mathtt{m}$ bits (or field elements, depending on context).

AsynchroMix proceeds in sequential mixing epochs, where in each epoch we mix input messages provided by $k \leq k_{pop}$ clients. Fig. 2.2 offers a high-level overview of the process. The protocol satisfies the following security properties:

- **Anonymity (Safety):** During every mixing epoch, even when all but 2 selected clients are compromised, the adversary cannot link an included message m_j to its honest client C_j except with probability negligibly better than 1/2.

  Specifically, for input vector $m_1, ..., m_k$ from k clients, the output is a permutation $\boldsymbol{\pi}(m_1, ..., m_k)$ such that the output permutation is at least almost independent of the input permutation.

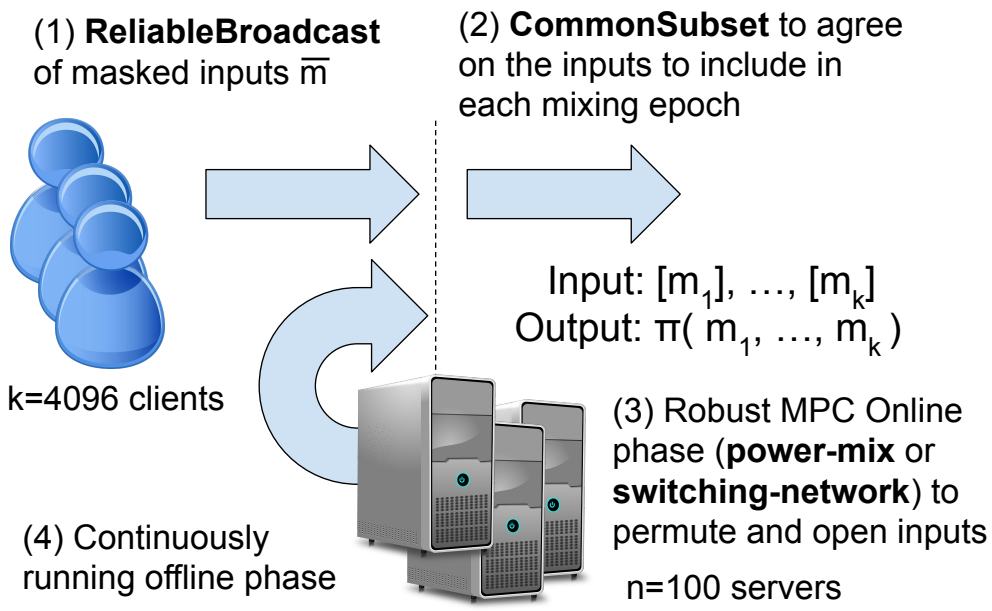

Figure 2.2. Overview of the AsynchroMix protocol [15]

- **Availability (Liveness):** Every honest client's input is eventually included in a mixing epoch, and every mixing epoch eventually terminates.

AsynchroMix is built upon a new MPC prototype, called HoneyBadgerMPC, which realizes secure computation through the use of asynchronous and maliciously-secure primitives. In particular, HoneyBadgerMPC makes use of asynchronous reliable broadcast to receive secret shared inputs from untrusted clients, and asynchronous common subset to reach agreement on the subset of clients whose inputs are ready and should be mixed in the next epoch. Each mixing epoch involves a standard robust MPC online phase based on Beaver triples and batched public reconstruction [9]. The offline phase [9], [23] runs continuously to replenish a buffer of preprocessing elements used by the online phase. The offline phase is optimistic in the sense that all server nodes must be online and functioning to replenish the buffer. These components are described in more detail below and illustrated overall in Figure 2.2.

2.2.1 Adversary Setting

We assume that at most $t < n/3$ of the servers are Byzantine corrupted by a global adversary, and similarly, any number of clients are corrupted as well.

2.3 Anonymous Communication Protocol I: Switching Network

Our first approach is to use an MPC program to randomly permute a set of k secret shared values using a switching network.

Switching networks are implemented in layers, where each layer applies a permutation to the inputs by conditionally swapping each pair. However, the resulting permutations are biased [48], [49]. For example, while a random Benes network can express every possible permutation, some permutations are more likely than others. Czumaj and Vöcking showed that $O(\log k)$ iterations of random butterfly networks (each of which consists of $O(\log k)$ layers) provide adequate shuffling [50] in the sense that the combined permutation is nearly uniform. The round complexity of the switching network is $O(\log^2 k)$, and the overall communication cost is $O(k \log^2 kn)$ considering there are $O(\log^2 k)$ layers in total and $O(k)$ multiplications are needed in each layer. Computation cost is $O(k \log^2 kn)$ since $O(k \log^2 kn)$ multiplications are needed in total. (See Figure 2.3 for a secure switching network instantiation with standard MPC operations.)

2.4 Anonymous Communication Protocol II: PowerMixing

In contrast with the switching network, we propose a novel protocol PowerMix, which results in reduced communication at the cost of computation. Our approach follows two steps. First, we compute the k powers of each shared secret, $[m^2], \ldots, [m^k]$ from just $[m]$. Surprisingly, we show how to achieve this using only $O(1)$ communication per shared secret, our protocol for computing powers may be of independent interest. The second step, inspired by Ruffing et al. [51], is to to use Newton's Identities [52] to solve a system of equations of the form $S_\mathrm{i} = m_1^\mathrm{i} + \ldots + m_k^\mathrm{i}$.

MPC Program switch

- Input : $[\mathrm{i}_1], [\mathrm{i}_2]$
- Output:$[o_1], [o_2]$ which are i_1 and i_2 swapped with $1/2$ probability
- Preprocessing: random bit $[b]$, $b \in \{-1, 1\}$
- Procedure:

  $[c] := [b] \cdot ([\mathrm{i}_1] - [\mathrm{i}_2])$

  $[o_1] := 2^{-1}([\mathrm{i}_1] + [\mathrm{i}_2] - [c])$

  $[o_2] := 2^{-1}([\mathrm{i}_1] + [\mathrm{i}_2] + [c])$

MPC Program switching-network

- Input : $[m_1], \ldots, [m_k]$
- Output:$\boldsymbol{\pi}(m_1, \ldots, m_k)$ where $\boldsymbol{\pi} \leftarrow \mathcal{D}$
- Procedure:
  - for each of $\log^2 k$ iterations, evaluate a switch layer, that uses k calls to switch to randomly permute all $k/2$ pairs of inputs, where the arrangement of pairs is laid out as $\log k$ iterations of a butterfly permutation
  - finally, reconstruct the output of the final layer, $\mathsf{Open}(\boldsymbol{\pi}([m_1], \ldots, [m_k]))$

Figure 2.3. Permutation based on a switching network

The servers can obtain S_i by computing locally $[S_\mathrm{i}]$ and publicly reconstructing. Then we solve the system of equations to obtain $\{m'_\mathrm{i}\}$ in canonical ordering. We next describe this approach in more detail.

2.4.1 Computing powers with constant communication

For each secret share $[m]$ sent by clients, we need to compute $[m^2], [m^3], \ldots, [m^k]$. The naïve way is to directly use Beaver triples $k-1$ times. If we care only about round complexity, we could also use the constant-round unbounded fan-in multiplication [53], though it adds a 3x factor of additional work. In either case, we'd need to reconstruct $O(k)$ elements in total.

We instead make use of a preprocessing step to compute all of $[m^2], [m^3], \ldots, [m^k]$ by publicly reconstructing only a single element. Our approach makes use of precomputed powers of a random element, $[r], [r^2], ..., [r^k]$ obtained from the preprocessing phase. We start with the standard factoring rule

$$m^k - r^k = (m - r)\left(\sum_{\ell=0}^{k-1} m^{k-1-\ell} r^{\ell}\right).$$

Taking $C = (m - r)$, and annotating with secret share brackets, we can obtain an expression for any term $[m^{\mathrm{i}} r^{\mathrm{j}}]$ as a sum of monomials of smaller degree,

$$[m^{\mathrm{i}} r^{\mathrm{j}}] = [r^{\mathrm{i}+\mathrm{j}}] + C\left(\sum_{\ell=0}^{\mathrm{i}-1} [m^{\mathrm{i}-1-\ell} r^{\mathrm{j}+\ell}]\right). \tag{2.1}$$

Based on Equation (2.1), in Figure 2.4, we give pseudocode for an efficient algorithm to output all the powers $[m^2], ..., [m^k]$ by memoizing the terms $[m^{\mathrm{i}} r^{\mathrm{j}}]$. The algorithm requires a total of $k^2/2$ multiplications and k^2 additions in the field. The memory requirement for the table can be reduced to $O(k)$ by noticing that when we compute $[m^{\mathrm{i}} r^{\mathrm{j}}]$, we only need monomials of degree $\mathrm{i}+\mathrm{j}-1$, so we can forget the terms of lower degree. Table 2.2 summarizes the asymptotic communication and computation costs of each approach.

Table 2.2. Summary of Online Phase computation and communication cost overhead (per client input) for Iterated Butterfly and PowerMix MPC programs

Protocol	Rounds	Comm. complexity	Compute
PowerMix	**2**	$O(n)$	$O(n + k^2)$
Switching Network	$\log^2 k$	$O(n \log^2 k)$	$O(n \log^2 k)$

2.4.2 Solving Newton's Identity

We now discuss how to reconstruct the shuffled values from the power sums. We have $S_{\mathrm{j}} = \sum_{\mathrm{i}=1}^{k} m_{\mathrm{i}}^{\mathrm{j}}$ where m_{i} is the message provided by client C_{i}. So we require an algorithm to extract the message m_{i} from S_{i}.

MPC Program compute-powers

- Input: $[m]$
- Output:$[m^2], [m^3] \ldots [m^k]$
- Precompute: k powers of random b, $[b], [b^2], [b^3] \ldots [b^k]$
- Procedure:

  Initialize $\mathsf{Array}[k+1][k+1]$

  for i from 1 to k: $\mathsf{Array}[0][\mathrm{i}] := [b^{\mathrm{i}}]$

  $C := \mathsf{Open}([m] - [b])$

  for ℓ from 1 to k: // compute all Array[i][j] where $\ell = \mathrm{i} + \mathrm{j}$

    $\mathsf{sum} := 0$

    for i from 1 to $(\ell - 1)$, $\mathrm{j} = \ell - \mathrm{i}$:

      sum += $\mathsf{Array}[\mathrm{i}-1][\mathrm{j}]$

      // Invariant: $\mathsf{sum} = \sum_{k<\mathrm{i}} [m^{\mathrm{i}-1-k} b^{\mathrm{j}+k}]$

      $\mathsf{Array}[\mathrm{i}][\mathrm{j}] = [b^{\mathrm{i}+\mathrm{j}}] + C \cdot \mathsf{sum}$

      // Invariant: $\mathsf{Array}[\mathrm{i}][\mathrm{j}]$ will store $[m^{\mathrm{i}} b^{\mathrm{j}}]$ by (2.1)

  for i from 2 to k output $[m^{\mathrm{i}}] := \mathsf{Array}[\mathrm{i}][0]$

Figure 2.4. Algorithm for calculating k powers of input $[m]$ using preprocessing in the Powermix online phase

Assuming that our goal is to mix k messages $m_1, m_2, m_3, \ldots, m_k$, the servers first run Algorithm 2.4 to compute the appropriate powers. Then all servers calculate $[S_{\mathrm{j}}] = \sum_{\mathrm{i}=1}^{k} [m_{\mathrm{i}}^{\mathrm{j}}]$ locally and then publicly reconstruct each S_{j}.

Let $f(x) = a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0$ be a polynomial such that $f(x) = 0$ has roots $m_1, m_2, m_3, \ldots, m_k$. And we have $a_k = 1$ given that it is the coefficient of x^k resulting from the product of $(x - m_1)(x - m_2) \ldots (x - m_k)$. According to Newton's identities [51], we can calculate all coefficients of $f(x)$ by:

$S_1 + a_{k-1} = 0$

$S_2 + a_{k-1} S_1 + 2a_{k-2} = 0$

MPC Program power-mix

- Input: $[m_1], [m_2], \ldots, [m_k]$,
- Output: a shuffling of $(m_1, m_2, \ldots, m_k)$
- Precompute: k sets of precomputed powers, for k instances of compute-powers (i.e., $[b_{\mathrm{i}}^{\mathrm{j}}]$ for $\mathrm{i} \in [1..k], \mathrm{j} \in [1..k]$,
  k beaver triples
- Procedure:

- *(Step 1)* for i from 1 to k:

  Run compute-powers (Algorithm 2.4) on $[m_{\mathrm{i}}]$ to obtain $[m_{\mathrm{i}}^2], [m_{\mathrm{i}}^3], \ldots, [m_{\mathrm{i}}^k]$

- *(Step 2)* for j from 1 to k:

  Locally compute $[S_{\mathrm{j}}] := \sum_{\mathrm{i}=1}^{k} [m_{\mathrm{i}}^{\mathrm{j}}]$

  $S_{\mathrm{i}} := \mathsf{Open}([S_{\mathrm{j}}])$

- *(Step 3)* Apply Newton's identities to solve $(S_1, S_2, \ldots, S_k)$, recovering a shuffling of $(m_1, m_2, \ldots, m_k)$.

Figure 2.5. Power-mixing protocol for shuffling and open secret shared values $[m_1], \ldots, [m_k]$

$$S_3 + a_{k-1}S_2 + a_{k-2}S_1 + 3a_{k-3} = 0$$

$\ldots$

Knowing S_{i} we can recover all a_{i} by solving these equations one by one. Once we know the coefficients of $f(x)$ we can then find k roots of $f(x) = 0$ with $O(k^2)$ computation complexity in our implementation [54]. Then we recover all m_{i}. Our final mixing set consists of these k messages.

To conclude, Figure 2.5 shows the overall protocol of PowerMixing.

2.5 AsynchroMix Offline Phase Requirements

The offline phase supporting AsynchroMix needs to generate the requisite preprocessing elements for both converting client inputs into secret sharings and for realizing either mixing

program. Of these, handling client inputs is the most straightforward as it only requires generating a t-shared random value for each input. For simplicity, we note that the randomness extraction protocol is just RanDouSha, but with only one matrix operation performed and with half the number of inputs and outputs. We, therefore, write randomness extraction as simply half of a call to RanDouSha. The details about the algorithm RanDouSha and BatchRecPub can be found in [23].

Running our mixing programs requires additional preprocessing inputs. The Switching-Network program requires the generation of random selector bits as well as the Beaver triples needed to use them. Meanwhile, our PowerMix program needs k secret-shared powers of the same random value. These preprocessing costs are given in terms of invocations of RanDouSha and BatchRecPub in Table 2.3.

Table 2.3. Offline phase requirements to run AsynchroMix $t + 1$ times

Preprocess Task	RanDouSha	BatchRecPub	Needed for
Client input:			
random $[r]$	0.5	1	each input
Switch Network:			
beaver triple	2	1	each switch
random bit $[b]$	1.5	1	each switch
Total:	$1.75k \log^2 k$	$k \log^2 k$	each epoch
PowerMix:			
k-powers	k	k	each input
Total:	k^2	k^2	each epoch

2.6 Supporting Larger Messages

We have so far assumed that each client message consists of a single field element (32 bytes in our concrete implementation), but AsynchroMix can easily be adapted to support larger (fixed-size) messages of multiple field elements each. Since the switching network choices depend only on the preprocessed selection bits, we can simply apply the same selection bits to each portion of input (i.e., the 1st element of clients' messages are permuted in the same way as the 2nd element, and so on). For PowerMix, we could reserve a portion of each message element (e.g., $\kappa = 40$ bits) to use as a tag which would be used to link parts of a message

together. Since no information about mixing inputs is leaked until the mix is opened, tags will not collide except for with $2^{-\kappa}$ probability.

2.7 Comparing AsynchroMix with Other Strong Anonymity Solutions

By the time this work is published, we observe that most anonymous communication systems do not focus on robustness and thus cannot achieve strong availability guarantees in the presence of faults. For example, in protocols following mix-nets strategies such as [31], [32], [55]–[57], nodes encrypt/decrypt layers of encryptions of user/cover traffic or re-encrypt batches of messages, and many failures have to result in users resending their messages. Similarly, in protocols following DC-net strategies such as [51], [58], nodes collaborate to randomly permute a set of messages while decrypting those, and any participating node may abort the execution and force re-execution. For these protocols to handle failures, it is necessary to rely on synchronous network assumptions to timeout a node, potentially restarting a computation or requiring users to resend messages. This introduces many potential issues. The first is that compromised nodes may attempt to degrade performance, such as by stalling until the last moment before being timed out. Attempting to optimize the protocol for speed by reducing the timeouts would only make it more likely that honest participants who experience a fault would be removed, thus degrading security. More importantly, by DoSing some honest nodes during re-running, it is also possible to launch inference attacks leading to deanonymization [51], [59], [60]. On the other hand, most of these schemes can indeed maintain anonymity/privacy against much larger collusion among the nodes, while liveness requirements of AsynchroMix in the asynchronous setting mandate us to restrict the adversarial collusions to $t < n/3$ nodes.

Our approach to MPC mixing is closely related to MCMix [29], which implements an anonymous messaging system based on MPC. Instead of a switching network, they associate each message with a random tag and obliviously sort the tags using MPC comparison operations.

There are also plenty of related works after AsynchroMix is published. Works such as Blinder [61], Clarion [62], and RPM achieved better performances through different methods, and we will cover them in the following chapters.

2.8 Security Analysis

Theorem 2.8.1. *Assuming that sufficient preprocessing elements are available from a previously-completed offline phase, then the AsynchroMix protocol satisfies the anonymity and availability properties defined earlier.*

Proof. For anonymity, it is clear that each mixing epoch only proceeds with k inputs from different clients. The use of preprocessed random sharings ensures that the secret shared inputs depend only on broadcast values from clients, and hence are valid sharings. The PowerMix program, thanks to perfect symmetry in its equation format, outputs the k values in a canonical ordering that depends only on their values, *not* their input permutation order. The Switching-Network induces a random permutation, which is sampled from a nearly uniform distribution.

For availability, we need to show that a) each honest client's input is eventually included in a mixing epoch, and that b) each mixing epoch completes robustly. For a), notice that once a broadcast $\overline{m}_{\mathrm{j}}$ from client C_{j} is received by every honest server, then the corresponding bits $b_{\mathrm{i,j}}$ in the next epoch will be set for every honest server. Therefore m_{j} is guaranteed to be included in the next mixing epoch. For b), notice that if at least $t+1$ of the bits $b_{\cdot,\mathrm{j}}$ are set for C_{j}, then we know at least one honest server has received the client's broadcast, and hence by the agreement property of ReliableBroadcast we can rely on this input to be available to every honest server. □

2.9 Robustness in MPC: HoneybadgerMPC

AsynchroMix is built upon a new MPC prototype, called HoneyBadgerMPC, which realizes secure computation through the use of asynchronous and maliciously-secure primitives. In particular, HoneyBadgerMPC makes use of asynchronous reliable broadcast to receive secret shared inputs from untrusted clients, and asynchronous common subset to reach agreement

on the subset of clients whose inputs are ready and should be mixed in the next epoch. Each mixing epoch involves a standard robust MPC online phase based on Beaver triples and batched public reconstruction [9]. The offline phase [9], [23] runs continuously to replenish a buffer of preprocessing elements used by the online phase. The offline phase is optimistic in the sense that all server nodes must be online and functioning to replenish the buffer.

2.10 Implementation and Evaluation

2.10.1 Implementation

We have developed a prototype implementation that includes all of the protocols needed to realize both the offline and online phases of AsynchroMix. Our prototype is written primarily in Python 3, although with computation modules written in C++ (to use NTL [63]).[2] For batch computations on secret sharings, both the FFT-based and matrix-based algorithms are implemented in C++ using the NTL library. We carried out a distributed benchmarking experiment with several aims: to validate our analysis, to demonstrate the practicality of our approach, and to identify bottlenecks to guide future improvement. We are mainly concerned with two performance characteristics: cost and latency. Latency is the user-facing cost, the time from when the user initiates a message to when the message is published. Computation and bandwidth costs are a complementary metric since we can improve latency by adding more resources, up to the extent that sequential computations and communication round trips are unavoidable. We are mainly interested in configurations with varying the mix size k, as well as the number of servers n (and assuming $n \approx 3t + 1$). We evaluated not only the *online phase* of the MPC protocols, but also the *offline phase* which generates precomputed Beaver triples, powers, and bits.

2.10.2 Performance Evaluation

To evaluate the performance of AsynchroMix and identify the tradeoffs and bottlenecks involved in our two mixing approaches, we deployed our prototype on clusters of AWS `t2.medium` instances (2 cores and 4GB RAM) in 10 regions across 5 continents. We conducted

[2] ↑https://github.com/initc3/HoneyBadgerMPC

baseline tests for bandwidth and latency between instances in different regions. For each experiment, we ran three trials for each configuration of n and k, and recorded the bandwidth, and running times.

Online Phase for PowerMix

Figure 2.6 (solid lines) shows the running time for PowerMix to mix and open from $k = 64$ to $k = 1024$ messages on up to $n = 100$ server nodes. It takes around 5 seconds to mix $k = 256$ messages on $n = 100$ servers and around 130 seconds to mix $k = 1024$ messages. We can see that PowerMix is mostly insensitive to the size of n, since the bottleneck is the computational costs, which depend mostly on k. Besides the computation steps could be parallelized to make use of more computation resources.

Figure 2.7 shows the communication cost of PowerMix, measured as outgoing bytes sent by each server, amortized per each client input. Since PowerMix requires two batch reconstructions of k shares each, and BatchRecPub has a linear asymptotic communication overhead to open a linear number of shares, we expect the per-server per-share cost to reach a constant for large enough n and k. We estimate this constant (the dashed line in the figure) as $2 \cdot 6 \cdot 1.06 \approx 12\times$, where the 2 is for the two batch reconstruction instances used in PowerMix, and the 6 is the overhead for each batch reconstruction (the limit approached by $\frac{2n}{t+1}$), and 1.06 is the observed overhead of Python `pickle` serialization in our implementation. As n grows larger, since there is an additive overhead quadratic in n, larger values of k are necessary for the amortization to have an effect. However, even at $n = 100$, only around 400 bytes are needed to mix each 32-byte message with $k = 512$ or higher.

Online Phase for Switching Network

Figure 2.6 (dashed lines) shows the running time for Switching Network to mix from $k = 64$ to 4096 messages. We can shuffle $k = 4096$ messages on $n = 100$ servers in around 2 minutes. Since the number of batch reconstruction rounds grows with $\log^2 k$, the sensitivity to n also increases as k increases.

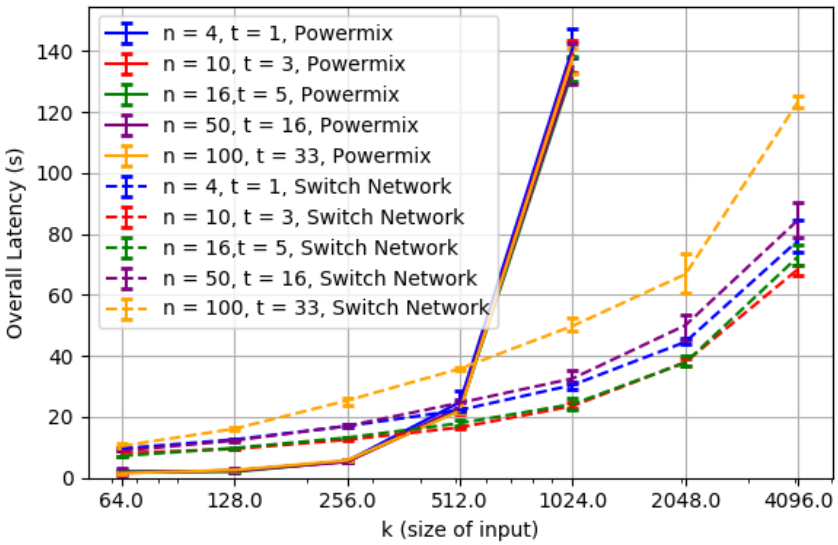

Figure 2.6. Online phase latency for varying number of client inputs, using PowerMix or Switching Network.

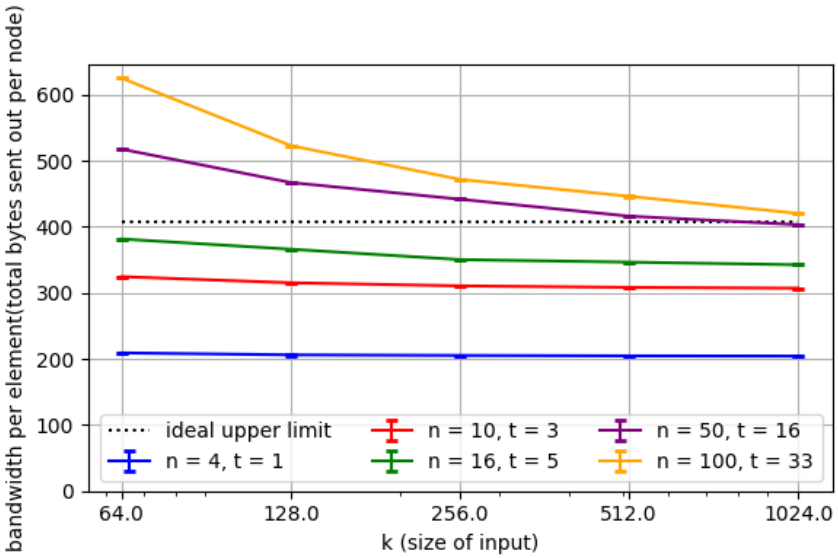

Figure 2.7. Communication cost (per node) of PowerMix in distributed experiment. Dashed line indicates the predicted limit as $\frac{2n}{t+1}$ approaches 6.

Based on the microbenchmarks, at $k = 4096$ and $n = 100$, the inherent computation time should account for only about 3 seconds out of the total 120 seconds observed. The rest is due to a combination of serialization and Python overhead as well as communication.

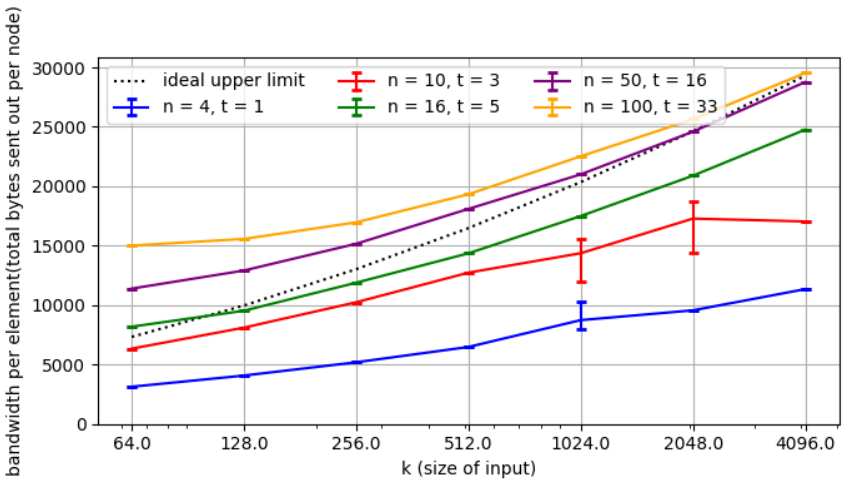

Figure 2.8. Communication cost (per node) of switching network in distributed experiment. Dashed line indicates the predicted limit as $\frac{2N}{t+1}$ approaches 6.

Fig 2.8 shows the overall communication cost of the Switching network. For $k = 4096$ client inputs with $n = 100$ servers, each input requires each server to transmit nearly 30 kilobytes. The dashed line here is $y = 32 \cdot 6 \cdot \log^2 k$ where 6 is reconstruction overhead and $\log^2 k$ corresponds to the number of total rounds. From our baseline experiment, the worst per-instance bandwidth is 221Mbps (São Paolo) and the longest round trip latency is 328ms (São Paolo to Mumbai), hence up to 50 seconds can be explained by transmission time and latency. Hence in this setting, computation, and communication contribute about equally (neither is the sole bottleneck), although there appears to a considerable room to eliminate overhead due to serialization and Python function calls in our implementation.

Overall cost for AsynchroMix

Figures 2.9 and 2.10 show the estimated overall cost, per server and per client input, combining both computation ($0.05 per core hour for an EC2 node) and bandwidth ($0.02 per gigabyte transferred out) costs based on AWS prices. The stacked bar charts show the costs broken down by phases (offline, online, and client input). The offline phase contributions are based on a distributed experiment for the RanDouSha algorithm, multiplied out by the necessary number of preprocessing ingredients of each type. The offline cost of PowerMix

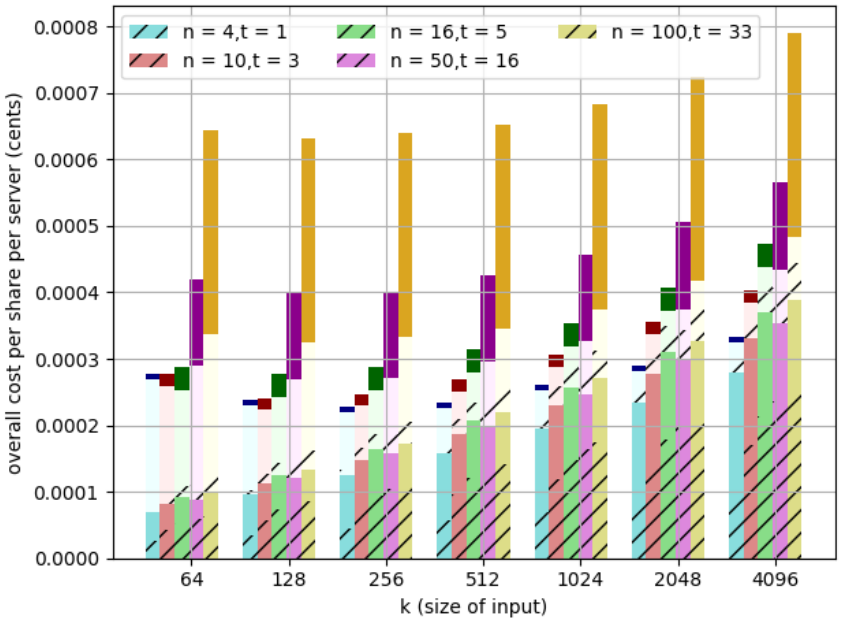

Figure 2.9. Estimated combined cost (computation and bandwidth) for AsyncMix with Switching Network. The cost includes offline phase cost(dark colored), online cost(light colored), and client input cost(top). Bandwidth cost is marked as "//".

is always more expensive than Switching Network at the same setting, and the difference increases with more clients (k versus $\log^2 k$). ***Using Switching Network, at*** $n = 100$ ***and*** $k = 4096$***, the overall cost (including all*** 100 ***servers) is 0.08 cents per message using geographically distributed*** `t2.medium` ***instances.***

2.11 Concluding Remarks

Emerging Internet-scale applications such as blockchains and cryptocurrencies demand a robust anonymous communication service offering strong security guarantees. Along the way towards building a robust anonymous communication service on top of MPC, we have highlighted robustness as a first-class concern for practical MPC implementations. Using an existing MPC implementation means accepting an *unfair* computation, which can enable intersection attacks when used for asynchronous communication. Furthermore, even a single faulty node could disrupt the service. Fortunately, we have shown through our AsynchroMix

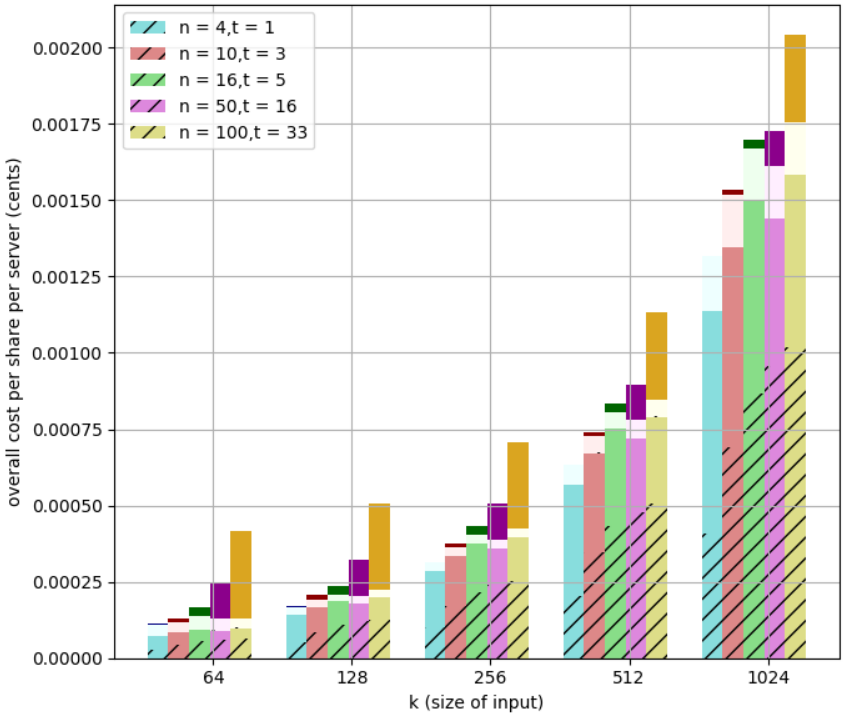

Figure 2.10. Estimated combined cost (computation and bandwidth) for AsyncMix with PowerMix. The cost includes offline phase cost(dark colored), online cost(light colored), and client input cost(top). Bandwidth cost is marked as "//".

application case study that robust MPC can be practical. Whereas related work explicitly foregoes robustness, we show that it is an achievable goal that is worth paying for.

AsynchroMix features a novel MPC program for anonymous broadcast that trades off local computation for reduced communication latency, allowing for low-latency message mixing in varying settings. Through an extensive experimental evaluation, we demonstrate that our approach not only leverages the computation and communication infrastructure available for MPC but also offers directions towards further reducing the latency overhead.

3. RPM: SECURE MIXING THROUGH PERMUTATION MATRIX

This work is a follow-up work of AsynchroMix [15]. As shown in the benchmark of AsynchroMix, the online performance of anonymous communication decreases significantly when the number of input messages increases. The main reason is that the computation complexity of the online phase of the PowerMix protocol is $O(k^3)$ for mixing k input messages. Therefore, the goal of this follow-up work is to design an MPC solution that is more scalable and efficient. In this chapter, we introduce RPM, an efficient MPC anonymous communication solution based on random permutation matrices.

3.1 Motivation and Introduction of RPM

There are by now millions of users using the Tor network [64], [65] to break the link between their identities and their messages/packets. As the solutions like Tor network suffer from attacks such as traffic analysis [66]–[68], anonymous communication becomes an active research area and many works [15], [31], [57], [61], [62], [69]–[71] aim at providing anonymous communication services efficiently. In this work, we explore the solutions of anonymous communication with the help of secure multi-party computation (MPC). MPC allows multiple distrusting parties to compute some functions collaboratively with their private input, thus it is a natural approach to building applications with robust privacy guarantees [15], [61]. This work proposes RPM[1] as a solution for anonymous communication in a client-server setting, where clients send their messages to the servers in a secret-shared manner, the servers randomly shuffle the messages and then output them to designated parties. Compared with existing works, the highlight of our protocols is that the clients only need the minimum cost to send the messages, meanwhile, the servers can perform the random shuffle very efficiently in terms of time and communication. It makes our protocol a preferred choice for real-time anonymous communication applications such as front-running-resistant market maker [72], [73]. Meanwhile, clients with limited computation power can benefit from our protocol significantly.

[1] ↑RPM stands for Random permutation matrix.

RPM makes use of the standard offline/online model of MPC [6], [15], [23], [74], [75], where the offline phase is used to prepare input-independent data, such that they can be consumed to accelerate the input-dependent online phase. The overall idea is that anonymous communication can be achieved by performing a random permutation to the input messages. Recall that for any k-input permutation $\boldsymbol{\pi}$, there exists a zero-one matrix $M_{\boldsymbol{\pi}}$ such that $\boldsymbol{\pi}(x) = M_{\boldsymbol{\pi}}x$. Therefore, if we obtain a permutation matrix such that anyone including the adversary has no knowledge about the underlying permutation, anonymous communication could be achieved by simply multiplying the matrix with the input vector, and it is equivalent to doing k inner product in parallel. In the offline/online model, we can generate a permutation matrix in the offline phase as it is input-independent, then achieve random permutation in the online phase through multiplications. This core idea is simple and fast, which are two great properties proven to be significantly useful in an anonymous broadcast scheme.

We provide three variants of the protocols, targeting different MPC frameworks and applications. The first variant leverages efficient inner product protocols to achieve a fast and cheap online phase. It only requires one communication round and k share reconstructions for mixing k messages. Then we present the second variant for MPC frameworks where an inner product protocol with constant communication complexity is not available. It requires 2 rounds and $2k$ reconstructions as a trade-off. Finally, we provide the third variant to handle a larger number of inputs more efficiently with cheaper offline phase cost and online computation complexity. As a result, all existing secret-sharing-based MPC frameworks can use our protocols for their own purposes. Besides, we show how to generalize our protocols to support more functionalities (e.g. two-way communication and anonymous messaging) and security properties (e.g. robustness, meaning that the protocols will make progress and finish with correct output even with the existence of malicious behaviors). What's more, as the core part of our protocols is the secure random permutation, it is of independent interest to more applications such as oblivious sorting and some graph-based algorithms.

We implement our protocols using MP-SPDZ framework [76], and benchmark both the online phases and the offline phases of all three variants of our protocols. These variants are implemented using different MPC back-ends provided by MP-SPDZ, which shows that our protocols can be used in most of the existing MPC frameworks. The results illustrate that

the first two variants of our protocol have great online&offline performances when dealing with a small number of messages (e.g. less than 10000). We can mix $k = 10000$ messages in around 0.58 seconds with 1.9MB communication. When dealing with a large volume of messages, our last variant achieves the best performance, which mixes $k = 160000$ messages in around 27 seconds with 88MB communication. The benchmark shows the offline phase is practical for real-world applications as well. Therefore, our protocol suits can handle different MPC frameworks and input volumes flexibly.

Finally, we modify the malicious secure back-end of MP-SPDZ to improve it from a secure-with-abort version to a robust version, which can be of independent interest for MPC applications. We test the performance of our protocols with a robustness guarantee, the benchmark shows that our protocols achieve robustness with no additional cost in the best cases, and around $2\times$ more time in the worst cases.

3.2 Related Works

The Tor network is a popular tool for anonymous communication; however, the current low-latency Tor design is significantly vulnerable to traffic analysis asymptotically [77] as well as empirically [66]–[68].

Mixing networks (mixnets) [78] improve the protection against traffic analysis through increased latency overhead in the form of communication over several hops (i.e., indirection) and mixing messages at one or more honest hops. Over the last four decades, numerous mix-net inspired protocols [31], [57], [70], [71], [79]–[82] have been proposed that can deter traffic analysis attacks; however, their high latency overheads of several seconds (at least) is unacceptable for many applications including browsing, messaging, or video calls. Moreover, mixnets are inherently non-robust as even a single node failure/crash can result in messages getting dropped.

For a high level of traffic analysis resistance while maintaining low latency, dining-cryptographers network (DC-net) [83] and its successors [84]–[90] are much better suited. Using a cryptographic setup/coordination among clients, these schemes offer provably strong

anonymity in a constant number of rounds [91]. However, as the number of clients grows client coordination can become an Achilles' heel for these DC-net-based solutions.

It is easy to observe that these DC-net systems are just types of MPC among the clients. Towards avoiding client coordination and expensive computation at the client-side, the idea of employing some MPC servers is getting popular [15], [61], [69], [92], [93]: here, similar to mixnets, every client is unaware of other clients and only communicates with the MPC servers. MPC servers perform some MPC protocols towards making clients' messages unlinkable to their identities. Among these MPC-based solutions, we find the works of AsynchroMix [15], Blinder [61], and Clarion [62] to be the closest to our work.

AsynchroMix [15] proposes two MPC solutions for anonymous broadcast. One method is based on the switching network, where the MPC performs $log(k)$ iterations of switching networks to simulate an almost-random permutation for k input messages. The round complexity of this method is $O(log^2(k))$ and the communication complexity is $O(klog^2(k))$. In their second method (so-called PowerMix), the messages are encoded into a symmetric equation system, then the anonymous broadcast can be achieved by solving it. The challenge for this method is that for any input secret-shared message, its powers are required by the equation system, and this leads to $O(k^3)$ computation complexity in the online phase. Although the computation is usually not considered as the bottleneck of an MPC protocol, the benchmark shows that the computation time actually dominates when k is large. Compared with PowerMix, our method reduces the computation complexity of the online phase to be at most $O(k^2)$, making it a better choice for a large volume of inputs.

Blinder [61] achieves anonymous broadcast by accumulating client messages in a large matrix. To achieve that, each client secret shares a matrix to the servers where all elements are zero but one position. The non-zero position is used to store the secret-shared message. The servers add up all the matrices from the clients and reconstruct the sum matrix to recover the messages. Several optimizations are applied to reduce the communication cost and to deal with the collisions when multiple clients choose the same position. In some sense, they achieve a scalable and efficient online phase by pushing some of the computation to the client side. Compared with blinder, the communication and computation cost of the client is cheaper in our protocol by an order of $O(\sqrt{k})$. Therefore, our protocol fits better

when clients have limited computation powers. Besides Blinder, Riposte [92] uses similar approaches of offloading part of the computation to the client-side. It uses discrete point functions to help reduce the client communication costs and achieves the same client-side complexity as Blinder.

Eskandarian and Boneh propose a protocol called Clarion [62], which is communication-efficient to do anonymous broadcast. They propose constructions for both the three-server setting and n-server setting, and the communication cost of their protocol is $O(k\ell)$ where k is the number of messages and ℓ is the size of the message. Their protocol has $O(n)$ round complexity in the n-server setting as it is made up of pairwise share translation. Our protocols and Clarion provide different trade-offs and fit different scenarios. Theoretically, our protocol has better round complexity and Clarion has better computation complexity. Therefore, there are settings where our protocols perform better and vice versa. Besides, due to the modular design, our protocols can inherently support stronger security properties (e.g. censorship-resistance, robustness, and fairness) if built with robust MPC libraries. For instance, if we build our protocols using HoneybadgerMPC[15], we will get exactly the same security properties as Asynchromix. However, it is impossible to do so for Clarion.

There are also works focusing on specialized applications. Spectrum [94] is designed for a broadcasting system where broadcasters share files anonymously with many subscribers. Subscribers send dummy files to form cover traffics. The evaluation results illustrate that Spectrum achieves better performance for scenarios with small broadcasters and many subscribers. Compared with their settings, all clients in our protocols are treated as "broadcasters" who can send messages anonymously.

3.3 Preliminary

3.3.1 System Model

We consider a standard client-server MPC setting with a set of n servers $P_1, P_2, \ldots, P_n$ and a set of k clients $c_1, c_2, \ldots, c_k$ $(k \geq 2)$. We assume that the servers already have key pairs established to build private, authenticated channels between each other. Besides, we assume clients connect to all the servers via TLS.

The whole protocol is divided into three phases as shown in Figure 3.1: (1) clients send their messages to servers in a private manner (via secret sharing). (2) Servers perform MPC protocols to randomly permute the inputs. (3) Servers reconstruct the permuted inputs to be the output of the protocol. We assume the client messages are the field elements with the same length, which can be achieved through padding. Fixed-length messages are essential since otherwise the message can be easily linked to its sender through the message size. Similar to existing works [15], [61], we assume servers have agreed on the set of client messages included in each protocol round, which can be achieved through any Byzantine agreement protocol.

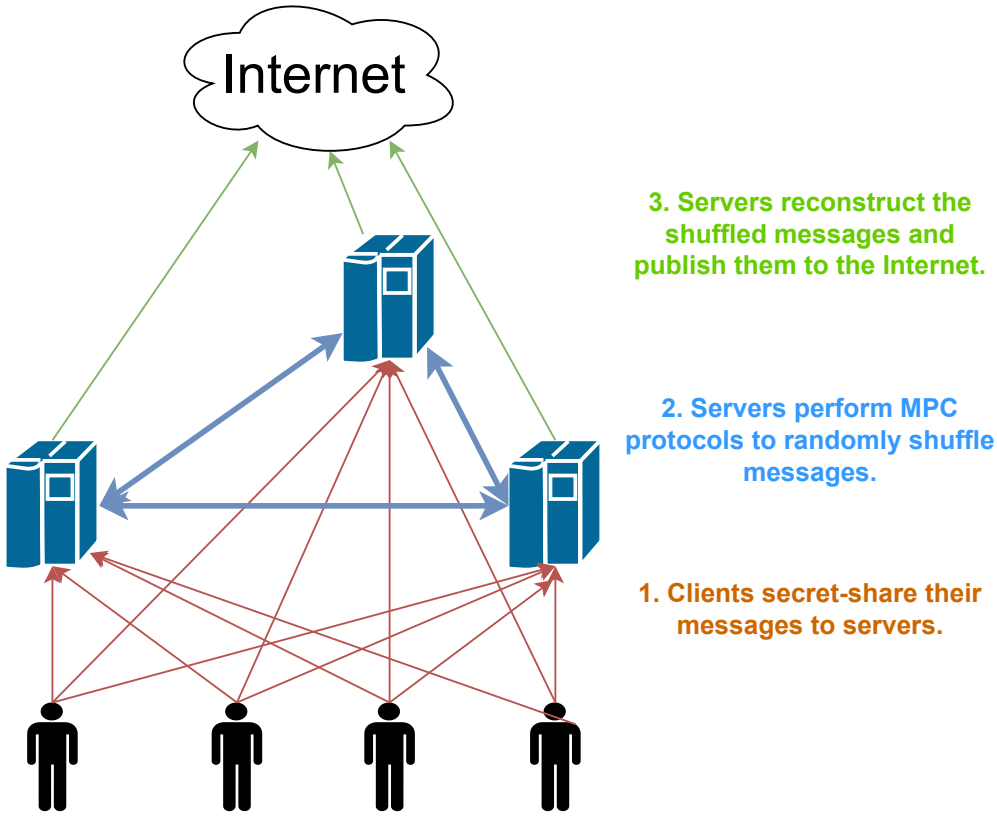

Figure 3.1. System Model of RPM

Since our protocol works across different communication settings, we do not put specific network assumptions such as partial-synchrony, bounded-synchrony, or asynchrony. Besides, the design goal of our protocol does not include protection against network-level attacks (e.g. DoS attacks).

The first variant of our protocol requires the usage of the Shamir secret sharing scheme or similar error-correcting code base secret sharing schemes. The second variant of our protocol gets rid of this restraint and can be applied to any secret-sharing-based MPC framework.

As for the adversary model, we assume there exists a static adversary that can corrupt at most t servers and at most $k-2$ clients. Our protocols are secure against a malicious adversary with $n \geq 2t+1$. In practice, our protocols can be implemented in any secret-sharing-based MPC framework, and the security of our protocols depends on the malicious secure building blocks of the underlying MPC frameworks. Besides, We propose verification checks to guarantee the malicious security of the offline phase. What's more, if the underlying MPC framework supports security properties such as guaranteed output delivery, our protocol should obtain those properties inherently.

3.3.2 Goals and Non-Goals

Below we list the goals that our protocols achieve:

Sender Anonymity: We want our protocol to achieve sender anonymity for the client message i.e., the ability of the adversary to figure out which client has sent a specific output message is no better than random guessing, even if all but two clients and any minority of servers are compromised.

Fast online phase: Our protocols lead to very efficient online phases in terms of communication, computation, and communication rounds, thus being good options for low-latency applications.

Light-weighted clients friendly: Our protocols require small communication and computation from the client-side.

Scalability: Our protocols can handle a medium volume of inputs within a short amount of time.

Then we list the non-goals below:

Confidentiality: Our protocols do not protect the confidentiality of the message content. Therefore, our protocols should be combined with other methods (e.g. encryption) to achieve confidentiality if it is required.

Network-layer Attacks: Similar to most existing works, our protocols are not designed to be resilient to network-layer attacks (e.g. DoS attacks).

Hiding Message Volume: Our protocols do not hide the global volumes of the messages.

3.3.3 Robust Secret Sharing Reconstruction

The reconstruction of Shamir Secret sharing could achieve robustness if robust polynomial interpolation is used. In this work, we use the idea of [15] to provide robust share reconstruction when it is required. The robust reconstruction requires $n \geq 3t + 1$ in a synchronous setting. We briefly introduce the construction below:

To reconstruct a secret robustly, the parties use the first $t + 1$ shares to reconstruct a polynomial ϕ, and use the rest t points to confirm all points correspond to the same polynomial. If any inconsistency occurs, the parties run the robust Reed-Solomon decoding with $3t + 1$ shares as the inputs. (If any share is missing, parties can use random values as the share and it will be treated as wrong shares and automatically corrected by the robust decoding algorithm). The procedure is described in Algorithm 1.

Algorithm 1: Robust Shamir share reconstruction

Input : $[S] = \{[s_1], \ldots, [s_n]\}$
Output : s

1 Interpolate a polynomial ϕ using any $t + 1$ shares.
2 Use another t share to check if they are generated using the same polynomial.
3 If it is true, output $s = \phi(0)$.
4 Else, run Reed-Solomon decoding with all the input shares to reconstruct ϕ‘, and output $s = \phi$‘(0).

The reason that the algorithm starts with a non-robust interpolation is that the non-robust interpolation is much cheaper compared with the robust version. If the non-robust interpolation succeeds, there is no need to run the expensive robust version. With this design,

if there are no malicious behaviors, the performance of the robust share reconstruction is the same as the non-robust version.

3.3.4 Notations

We summarize notations specific to RPM here. We denote $[s]$ as a secret sharing of the secret field element s. Besides, we use capital letters to represent matrices or vectors (S and $[S]$). We denote **Open**($[s]$) as the reconstruction of the secret share, and we use **Mul**($[x]$, $[y]$) to represent the Beaver Multiplication of two secret shares $[x]$ and $[y]$ or two secret-shared matrices/vectors $[X]$ and $[Y]$. We use **Inner-prodcut-and reconstruct**($[X]$, $[Y]$) to represent an algorithm that computes the dot product of two input vectors X and Y and reconstructs the results.

3.4 Using Permutation Matrices for Anonymous Communication

3.4.1 Overview of the Variants

We present three variants of our protocol targeting different MPC frameworks and applications. The first variant is designed for MPC frameworks with an efficient secure inner product protocol implemented [40], [95], [96] (i.e. each inner product can be evaluated with a constant number of reconstructions independent of the vector size). The second variant gets rid of the secure inner product in the online phase, with a cost of a little more expensive offline phase and one more round in the online phase. Therefore, the second variant fits better with MPC frameworks that do not support efficient secure inner product evaluations. The third variant is designed for a large number of inputs (e.g. $k > 10000$), as the offline phases of the first two variants take a long time when k is large. Besides, Variant 3 has cheaper online computation complexity, which we find is the bottleneck of the protocols for large k. As a trade-off, the third variant takes more online communication and rounds.

3.4.2 Collecting Client Messages

As the clients can be corrupted by a malicious adversary, the messages they share to the servers may not be a valid (n, t) secret sharing. To solve this problem, we use a similar method used in [15]: servers can prepare a random share $[r]$ for each input client message m. During the input phase, all servers send their shares of r to the client, such that the client can reconstruct r, and broadcast $m + r$ to servers, each server then compute their share of the message $[m] = m + r - [r]$. Since $[r]$ is guaranteed to be a valid (n, t) secret sharing, the share of the client input is guaranteed to be well-formed. The computation and communication required by clients are both $O(n)$.

3.4.3 Supporting Messages with Large Size

Our protocols can be easily adapted to handle large messages. If the message is too large to fit in one single field element, clients can divide the large messages into pieces with the same length and represent them using multiple field elements (padding may be required for the last block). In the online phase, the servers can use the same permutation matrix P to permute all the message pieces, such that the same permutation is performed on all client messages.

3.4.4 Malicious Security

To achieve malicious security, our protocols should be built on several malicious secure building blocks. More concretely, we require a malicious-secure secret sharing scheme [15], [20], [97], [98] to guarantee the correctness of the secret sharing reconstruction. We also require malicious-secure share multiplication to guarantee the correctness of matrix operations and vector operations. A malicious secure inner product protocol [23], [98] is required in one of our variants. We follow a modular design such that any building blocks achieving malicious security can fit our protocols. Besides, our protocol can also benefit from future building blocks with better efficiency in a plug-and-play manner.

3.4.5 The First Variant

Offline Phase

The goal of the offline phase is to generate a random permutation matrix such that the adversary has no information about the permutation. To achieve that, we ask $t+1$ servers to generate a random permutation matrix each, and secret-share them to all parties. Then all parties multiply these shared matrices together to get the final permutation matrix. Since there is at least one matrix provided by the honest server, the adversary has no knowledge about the final combined permutation. Note that this step requires the multiplication of $t+1$ matrices, so our offline phase is more suitable for the settings where the number of servers is small. Besides, for small k, we can use existing methods to efficiently evaluate the multiplication of multiple matrices such as [6].

In the malicious setting, we also need to guarantee that the matrices shared by servers are indeed the permutation matrices. Therefore, the following two checks have to be performed: (1) the elements of the matrix are either zero or one. (2) The weight of each row and each column is exactly one (i.e. Each row only has one position to be one, and all other positions are zero.). To finish these checks, we can use a linear sketch for the language of vectors of hamming weight one [61], [99], [100]. To verify a vector $w = (w_1, \ldots, w_k)$, the sketch is represent by $(\sum_{\mathrm{i}=1}^{k} w_{\mathrm{i}} \cdot r_{\mathrm{i}})^2 - m(\sum_{\mathrm{i}=1}^{k} w_{\mathrm{i}} r_{\mathrm{i}}^2)$ where r_{i} are public random values and m is the value in the single non-zero entry (in our case $m = 1$). If the vector w has a hamming weight greater than one, then the sketch outputs a non-zero value with probability $\frac{1}{\|F\|}$, where $\|F\|$ is the size of the ring or field.

We can apply this sketch to the final combined permutation matrix and both properties can be properly verified. The cost of this check is cheap since each sketch only includes one secret sharing multiplication. Therefore, the overall communication complexity is $O(k)$ and the round complexity is one. The offline phase is summarized in Algorithm 2.

Algorithm 2: The offline phase of Variant 1

for i $\leftarrow 1$ *to* $t+1$ **do**
  Server P_i generates a k-by-k permutation matrix M_i and secret-share it to all servers
All Servers multiply and compute $[P] = [M_1][M_2]\dots[M_{t+1}]$
for i $\leftarrow 1$ *to* k **do**
  All servers perform sketch checks mentioned in Section 3.4.5 on the i-th row and i-th column of $[P]$.
If any check fails, abort.
Otherwise, output $[P]$.

Online Phase

In the online phase, we can achieve permutation by simply multiplying the permutation matrix M and the input vector X. Considering X is a vector, this is essentially k dot products and they can be computed in parallel. The protocol is summarized in Algorithm 3.

There are existing works [23], [40], [96] showing how to do dot product efficiently. As an example, we show how degree-$2t$ polynomial interpolation can be used to efficiently compute and reconstruct the inner product of two secret shared vectors $X = \{x_1, x_2, \dots, x_k\}$ and $Y = \{y_1, y_2, \dots, y_k\}$:

In the online phase, parties locally compute $[x_i y_i]_{2t} = [x_i]_t \cdot [y_i]_t$ for all i, then they compute and reconstruct the inner product result $Z = \sum_{i=1}^{k} [x_i y_i]_{2t}$ by reconstructing a degree-$2t$ polynomial.

If we use the protocol above to compute all the inner products in parallel, the round complexity of our online phase is only one. The communication complexity is $O(k)$ as there are k reconstructions needed in total.

However, the inner product protocol introduced above has some constraints, therefore not all MPC frameworks support it naturally. For instance, this protocol only works with Shamir-secret sharing (or similar error-correcting-code-based secret sharing schemes), and does not work on schemes such as additive secret sharing or replicated secret sharing. Besides, as degree-$2t$ polynomial reconstruction is required, the protocol may need more portion of

parties to be honest (e.g. $n > 3t + 1$). To mitigate this problem, we design another variant of our protocol Variant 2, where the secure inner product is not required in the online phase.

Algorithm 3: The online phase of Variant 1 (P[i] denotes the i-th row of the matrix P)

Input : $[X] = \{[x_1], \ldots, [x_k]\}$
Output : Y
Pre-computation : Permutation matrix $[P]$

```
Y = {}
for i ← 1 to k do
    Y[i] = Inner-product-and-reconstruct([P[i]], [X])
Output Y.
```

3.4.6 The Second Variant

The design goal of this variant of our protocol is to get rid of the inner product in the online phase. To achieve that, we add an additional step in the offline phase such that the inner product computation is shifted into the offline phase. What's left for the online phase is simply some secret share reconstructions. The key observation of this protocol is an equation $PX = P(X + R) - PR$, where P is the permutation matrix, X is the input message vector, and R is a vector of random shares. We leverage R as a mask vector such that we can safely reconstruct $X + R$ in the online phase, and PR can be prepared in the offline phase as it is independent of the input X. Finally, the permutation result PX can be written as a linear combination of the secret shares above.

Offline Phase

The first part of the offline phase is still to generate a shared permutation matrix, and the steps are the same as Variant 1. After that, all parties collaboratively generate k random shares $[R] = \{[r_1], \ldots, [r_k]\}$, then they compute $[Y] = [P][R]$ through k inner products. Note that a more expensive inner product can be used here as it happens in the offline phase, and an inner product protocol with $O(k)$ reconstructions per random r is perfectly fine because it will not explode the complexity of the offline phase anyway, the offline phase complexity is

still bounded by the generation of the permutation matrix. We can think of this approach as shifting the inner product computation from the online phase to the offline phase with the help of some randomness R. Finally, all parties take $[P]$, $[R]$, and $[PR]$ as the output of the offline phase. The protocol is summarized in Algorithm 4.

Algorithm 4: The offline phase of Variant 2

for i $\leftarrow 1$ *to* $t+1$ **do**
  Server P_i generates a k-by-k permutation matrix M_i and secret-share it to all servers
All Servers multiply and compute $[P] = [M_1][M_2]\dots[M_{t+1}]$
for i $\leftarrow 1$ *to* k **do**
  All servers perform sketch checks on the i-th row and i-th column of $[P]$
If any check fails, abort.
All servers generate k random shares $[R] = \{[r_1], [r_2], \dots, [r_k]\}$.
All servers compute $[PR] = Mul([P], [R])$.
Output $[P]$, $[PR]$, $[R]$.

Online Phase

Given the input messages vector $X = \{x_1, \dots, x_k\}$ and the offline phase output, all parties compute and reconstruct $X + R$ in the first round. Then they can locally compute the share of the output as $[PX] = [P](X + R) - [PR]$, and reconstruct PX in the second round. As we mentioned, the secure inner product is no longer needed in this variant, and the cost is k more reconstructions and one more round. The protocol is summarized in Algorithm 5.

Algorithm 5: The online phase of Variant 2

Input : $[X] = \{[x_1], \dots, [x_k]\}$
Output : Y
Pre-computation : $[P], [R], [PR]$
$[X + R] = [X] + [R]$
$X + R = Open([X + R])$
$[Y] = [P] \cdot (X + R) - [PR]$
$Y = Open([Y])$

3.4.7 The Third Variant

The first two variants illustrate great performance when dealing with a small volume of inputs. However, the offline phase of the first two variants requires preparing k-by-k permutation matrices, thus the size of the matrices increases quadratically with k, and it becomes too huge to be used in practice for large k.

To solve this problem, we propose our third variant that performs much faster for larger k in both the online phase and the offline phase. The idea is based on the permutation network[69], [101]. In [101], Hastad analyzed the efficiency of random permutation using a *square network*. A square network for k inputs consists of q layers, where each layer consists of $\sqrt{k}$ of permutation nodes. Each permutation node takes $\sqrt{k}$ inputs and randomly permutes them, then sends the outputs to the next layers in a butterfly network fashion. We present an example of a square network in Fig. 3.2. The study of Hastad illustrates that this network can achieve a nearly random permutation after only $q \in O(1)$ iterations. (e.g. the result shows that after $q = 15$ layers, the outputs are close to a random permutation of inputs)

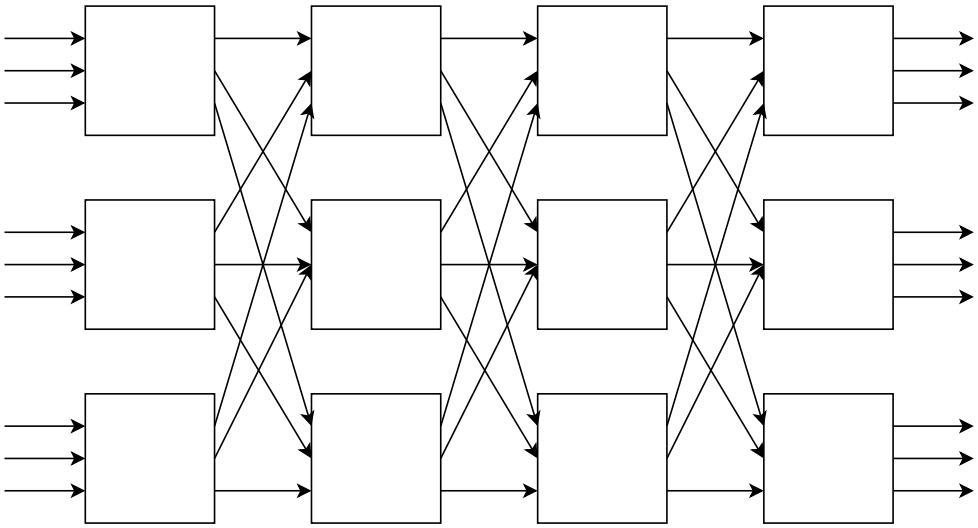

Figure 3.2. An illustrative example of a square network with $k = 9$. The network consists of $q = 4$ layers, where each layer has $\sqrt{k} = 3$ permutation nodes, represented by square blocks. Each permutation node takes $\sqrt{k} = 3$ messages as inputs, randomly permute and output them. In Variant 3, we can initialize each permutation node using either Variant 1 or Variant 2. All permutation nodes in the same layer can be executed in parallel.

Variant 3 implements a square network by realizing each permutation node with either Variant 1 or Variant 2. This significantly reduces the computation cost of the offline phase. In Variant 1 and Variant 2, the offline phase has to prepare a k-by-k permutation matrix. In Variant 3, the offline phase generates $q\sqrt{k}$ matrices of size $\sqrt{k}$-by-$\sqrt{k}$. Considering the matrix multiplication is required in the offline phase, this reduces the computation complexity of the offline phase from $O(k^3)$ to $O(q\sqrt{k} \cdot t\sqrt{k}^3) = O(k^2)$. Besides, the computation complexity of the online phase is also reduced by a factor of $\sqrt{k}$ because the vector size of each dot product is significantly reduced.

As a trade-off, the online phase requires higher but still constant rounds to finish. The parameter q is flexible and can be changed based on the time available for the offline phase and the anonymity strength. We pick $q = 15$ in our experiments for a strong anonymity guarantee.

3.4.8 Cost Analysis and Comparisons with Related Works

For Variant 1, the offline phase requires $t+1$ servers to each generate a k-by-k permutation matrix, and multiply them together. This requires $O(nk^3)$ local computation, $O(\log n)$ rounds and $O(nk^2)$ communication. The verification check for the malicious security takes one round and $O(k)$ communication. As for the online phase, since it is k dot products in parallel, the overall communication is $O(k)$ and the round complexity is one.

For Variant 2, the offline phase cost is the sum of the first variant offline phase cost and k dot product protocol. Consider a dot product protocol where parties simply use beaver triples to compute inner products, the communication complexity for k dot product of length-k vectors is $O(k^2)$. Therefore, the overall offline phase communication cost is $O(nk^2 + k^2) = O(nk^2)$. The online phase only consists of two rounds, where each round reconstructs k secrets.

For Variant 3, the offline phase requires $O(nk^2)$ local computation, $O(\log n)$ rounds and $O(nk^{1.5})$ communication. We then explain why the online phase is also the most efficient for large k: Our benchmark illustrates that the online phase is heavily bottlenecked by the local computation when k is large, which takes more than 95% of the overall running time. For the

first two variants, the computation complexity is $O(k^2)$ because both variants require k dot products between size-k vectors. For Variant 3, each layer includes $\sqrt{k}$ permutation nodes, with each node doing $\sqrt{k}$ dot products between size-$\sqrt{k}$ vectors. Since we have a constant number of layers, the overall computation complexity is $O(k^{1.5})$. Therefore, Variant 3 achieves the best performance on the main bottleneck when k is large. We highlight that when k is small, the first two variants could be better choices since the online phase is bottlenecked by the communication and rounds in those cases.

We summarize the theoretical online complexity of our work and related works in Table 3.1, the comparison shows that our protocol achieves the best server-server performance in the online phase, meanwhile keeping the client-server cost minimum. As for the offline phase cost, we do not provide a similar table as some protocols are unclear about their offline phase costs. Here we just compare the offline phase cost between our protocols and the PowerMix [15], which has the closest online phase communication and rounds. The communication cost of Powermix offline phase is $O(k^2)$ and the round complexity is $O(\log k)$. Meanwhile, our first two variants have offline communication cost $O(nk^2)$ in $O(\log n)$ rounds. Variant 3 has the cheapest offline phase computation cost by a factor of $O(\sqrt{k})$. We can observe the trade-off here: we achieve a better online performance than Powermix by pushing more computations to the offline phase. What's more, when the number of servers n is small such that it can be treated as a small constant, the offline cost of our protocol is in the same order of magnitude as PowerMix. In real-world applications (especially in MPC-as-a-service settings), a small number of servers are usually preferred considering the cost of setting up these expensive high-end machines.

3.4.9 Security Analysis

In what follows, we informally define the security properties we expect from our protocols:

Correctness: At the end of a successful run our protocol, all servers output a set of plaintext messages, which is a random permutation of all the input client messages.

Table 3.1. Comparison of the Online Phase Performance For Recent Anonymous Communication Protocols (n is the number of servers, where t of them can be corrupted. k is the number of client messages. q is the depth of the square network, which is a small constant. The server-server communication is measured by the number of secret sharing reconstructions required. The client-server communication is measured by the number of messages sent by a client to a single server.)

	Client-server communication	Client Computation	Server-server Communication	Server Computation	Server Rounds	Resilience	Robustness Capability
McMix [69]	$O(1)$	$O(1)$	$O(\alpha k \log k)$*	$O(\alpha k \log k)$*	$O(\log k)$*	$n = 3, t = 1$	✗
Switching Network [15]	$O(1)$	$O(n)$	$O(k \log^2 k)$	$O(k \log^2 k)$	$\log^2 k$	$n \geq 3t+1$	✓
PowerMix [15]	$O(1)$	$O(n)$	$O(k)$	$O(k^3)$	2	$n \geq 3t+1$	✓
Blinder [61]	$O(\sqrt{k})$	$O(n \cdot \sqrt{k})$	$O(k)$	$O(k^2)$	$O(1)$	$n \geq 4t+1$	✓
Clarion [62]	$O(1)$	$O(n)$	$O(k)$	$O(k)$	$O(n)$	$n > t$	✗
Variant 1	$O(1)$	$O(n)$	k	$O(k^2)$	1	$n \geq 2t+1$*	✓
Variant 2	$O(1)$	$O(n)$	$2k$	$O(k^2)$	2	$n \geq 2t+1$*	✓
Variant 3	$O(1)$	$O(n)$	$O(k)$	$O(k^{1.5})$	q	$n \geq 2t+1$*	✓

* $n \geq 2t+1$ is the default model setting for our malicious secure protocols (especially for Variant 1). Variant 2 can support $n > t$ if built in dishonest-majority MPC frameworks. Meanwhile, more restriction might be needed to support more security properties (e.g., $n \geq 3t+1$ for robustness).

* In McMix [69], α refers to the number of reconstructions needed for a single secure comparison protocol. McMix requires $O(k \log k)$ secure comparisons evaluated in $O(\log k)$ rounds.

Sender Anonymity: The ability of the adversary to figure out which client has sent a specific output message is no better than random guessing, even if all but two clients and any minority of servers are compromised.

The correctness of the protocol is trivial from the use of the permutation matrix. As long as the permutation matrix P is valid, the computation result is guaranteed to be a permutation of input vectors. The validity of the permutation matrix is verified in our offline phase through linear sketch checks in the malicious setting.

For sender anonymity, we can prove that the transcript of the adversary only includes unrelated random values, therefore it cannot differentiate any input messages.

We provide the general idea here and the intuition is that the transcript of our protocol includes the offline data, the reconstructed $X + R$ in Variant 2, and the final output. The offline data are all in the form of secret shares, and they are independent of the client inputs, thus the adversary has no information about either the plaintext offline data or the client inputs. The reconstructed $X+R$ is also random because R contains elements picked uniformly random from the field. What's more, the randomness of offline phase and the randomness of $X + R$ are independent of each other. To conclude, the transcript of our online protocol only contains unrelated random elements, therefore they are indistinguishable among one another.

For Variant 3, the protocol only invokes our first two variants multiple times, and the values outside of permutation nodes are all secret-shared. Therefore, the privacy of the third variant is reduced to the privacy of the first two variants.

We then discuss the security against a malicious adversary. In the offline phase, the adversary can submit an arbitrary matrix as permutation matrices, however, this will be captured by the sketch check. The adversary has no information about the combined permutation matrix because at least one random permutation is provided by an honest party. Therefore, the adversary cannot alter the protocols in the offline phase without being captured. In the online phase, our protocol simply invokes malicious secure building blocks, thus the security is reduced to the security of those building blocks. In Variant 1, the online phase only includes k malicious-secure inner product. In Variant 2, the online phase includes $2k$ malicious-secure share reconstruction.

3.5 Applications of RPM

We so far focused on using RPM to achieve anonymous broadcast. Below we show some higher-level applications with RPM as building blocks.

3.5.1 Two Way Communication

First, we show how to extend our protocols to support two-way communication, which allows the receivers to reply to the sender's messages anonymously. We notice that this feature is similar to anonymous messaging [61], [62], [69], where senders and receivers conduct private conversations such that the adversary has no information about their identities.

Two-way communication is split into two parts: In the first part, the sender sends its message anonymously. In the second part, the receiver recognizes the message from the sender and sends the reply message back to the sender. We can achieve the first part using any variant of our protocols, such that the output messages $Y = \pi(X)$ is a random permutation π of the input messages X. To help the receivers recognize the messages, senders and receivers agree on some tags offline, such that these tags can be prefixed to the sender messages, and the receivers can recognize the messages through the tags. As for the second part, the key observation is that the permutation we perform in the first part can be reversed through the inverse permutation π^{-1}, which is available by computing the inverse of the permutation matrix. Therefore, the receivers can put their reply messages in the same position as the sender message, then the servers do a second round of mixing protocols using the inverse of the permutation matrix π. Instead of reconstructing the permutation outputs publicly, the servers send their shares to the designated senders such that the senders can reconstruct the reply messages privately. In this way, the adversary has no information about the output of the second round of mixing, therefore cannot build any link between the sender messages and the reply messages. As for the computation of the inverse of a permutation matrix, we notice that it can be achieved by simply computing the transpose because it is an orthogonal matrix. The computation of the transpose is just a relocation of matrix elements and therefore is a local computation. An example of two-way communication is shown in Figure 3.3.

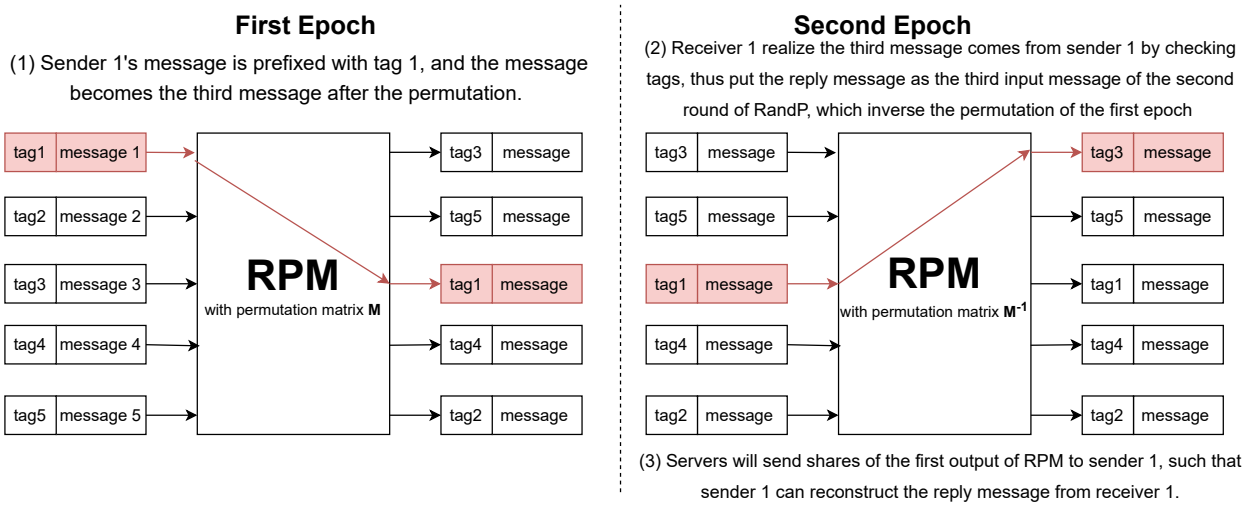

Figure 3.3. An example of two-way communication. The example includes five participating clients, and we mark the protocol flow for the first sender (denoted as sender 1) in red. Sender 1 and the corresponding receiver agree with a tag (tag1 in the figure) before the protocol.

If the application also requires hiding the content of the sender messages, some extra steps should be deployed on the sender side (e.g., senders can encrypt their messages except the tags using symmetric key encryption, share the key with the receiver offline, then send the encrypted messages to our protocols). In this case, our protocol achieves the same functionality as anonymous messaging [61], [69].

Since the design is simply two runs of our secure mixing protocols, it is secure against malicious servers naturally. As for the malicious clients, the worst case is that he/she can reply to a message not belonging to him/her. To avoid it, we require the sender and the receiver also agree on some randomness offline (e.g., a common string), such that the sender could hash the randomness, and put the hash of the randomness as the tag. The servers allow a receiver to reply to the message only if he/she can provide the common randomness that matches the hash value.

We observe that our protocol leaks the number of messages that receive replies. However, this information is not required to be hidden in most applications.

3.5.2 Secure Sorting

The secure sorting takes private inputs from k clients, and outputs the sorted inputs without revealing their ownership. There are in general two types of sorting algorithms. The first kind is data-dependent sorting, where the input decides the execution path of the algorithm. A good example is the quicksort, where the choice of pivot decides the number of recursions. Therefore, the execution path (e.g. execution time) leaks information about the input, and most existing works choose to implement the second type of sorting algorithm so-called oblivious sorting [102]–[104]. However, to the best of our knowledge, the most practical oblivious sorting is achieved by sorting networks [104] with $O(k \log^2 k)$ communication in $O(\log^2 k)$ rounds.

Recently, Hamada et, al [105] propose to use data-dependent algorithms in an oblivious fashion to solve the sorting problem. The idea is that parties can perform secure random shuffle to the input, reconstruct the inputs, then compute data-dependent algorithms locally with reconstructed input. With this idea, the secure sorting problem is reduced to secure random shuffle, which can be achieved through our random permutation protocol. By applying our protocol there, secure sorting can be achieved by only $O(k)$ communication in one or two rounds.

Secure sorting itself is a vital build block of various high-level applications such as secure auctions [106], [107], combinatorial graph problems [108], and network flow problems [109]. Therefore, our protocols can be beneficial to much more applications than just anonymous communication.

3.6 Implementation and Evaluation

3.6.1 Implementation

There are currently many MPC libraries [15], [40], [76], [95] available with different trade-offs. Among them, we choose to use MP-SPDZ [76] to implement all three variants of our protocols, because MP-SPDZ is a collection of multiple MPC back-ends, and it allows us to pick proper back-ends for different variants to get the best performances. Besides, it

helps us to illustrate that our protocols can suit almost all the MPC back-end because of the fact that only the basic building blocks are needed. The code is for now available in an anonymous link[2], and will be open-sourced in the full version of this paper.

As Variant 1 requires a fast malicious secure dot product protocol, the number of the back-ends satisfying the requirements is limited. Among them, we find the SY-SPDZ back-end to be the one with the best performance, thus choosing it for the benchmark. For Variant 2, the back-end we use is malicious-shamir-party of MP-SPDZ, as an efficient inner product is not required. For Variant 3, we built up each permutation node using Variant 2, thus using the same back-end. In general, we are interested to answer how fast our online protocol can be, as the protocols target real-time applications where the latency is the most significant. Therefore, in the experiments we mainly report the online running time and the online communication time. To simulate the real-world use cases, we run the experiment using Amazon AWS. Besides, we also conduct experiments to measure the cost of the offline phase to confirm it is practical.

The code is written in MP-SPDZ customized language. Variant 1 directly invokes the inner product protocols k times in parallel, then the results are reconstructed in the second round. Note that in our original protocol, the inner product and the reconstruction could be compressed into one single round, however, our implementation requires two rounds to fit the framework more easily. Variant 2 invokes the reconstruction protocols in two rounds, where each round reconstructs k secret shares. We notice that because of the limitation of the framework, it actually takes more than 2 rounds to finish the reconstruction when k is large. Besides, multi-threading is used with up to 32 threads when applicable to speed up the local computation.

3.6.2 Online Phase Evaluation

We run the benchmark on AWS EC2 clusters in a three-party malicious-secure setting. The AWS instance we use is c5.9xlarge with 32 cores and 72GB RAM. All three machines are in the same region (US.East).

[2]↑the anonymous link: https://anonymous.4open.science/r/MP-SPDZ-811B/

Variant 1

First, we present the benchmark result of Variant 1. The result is available in Table 3.2. It shows that our protocol can mix $k = 10000$ messages in around 1.5 seconds and 1.483MB communication. The communication cost increases linearly with the number of clients, which is consistent with our theoretical complexity. We are also interested in the bottleneck of this protocol, so we also implemented the non-multi-threading version of the protocol, its benchmark shows the local computation dominates over 95% of the overall running time. Therefore, the multi-threading significantly improves the performance of our protocols as we confirm local computation is the bottleneck especially when k is large.

Table 3.2. Performance of the online phase of Variant 1 in three party setting. (k refers to the number of clients. We assume each client sends a 16 byte field element as the message in each execution. Communication is measured by total MB sent per party)

k	Online Time (s)	Online Communication(MB)
1000	0.05	0.961
3000	0.132	1.077
5000	0.360	1.193
7000	0.689	1.309
10000	1.485	1.483

Variant 2

The benchmark numbers of Variant 2 is available in Table 3.3. Interestingly, the performance of Variant 2 is better than Variant 1, although theoretically Variant 1 should perform better. We think the reason is that the MPC back-end of Variant 2 is faster than Variant 1, although it does not support fast inner product. As a result, we also use Variant 2 as the building blocks to conduct the benchmark of Variant 3.

Table 3.3. Performance of the online phase of Variant 2 in three party setting. (k refers to the number of clients. message size is 16 bytes. Communication is measured by total MB sent by all parties)

k	Online Time (s)	Online Communication(MB)
1000	0.02	0.193
3000	0.066	0.577
5000	0.155	0.960
7000	0.288	1.345
10000	0.580	1.921

Variant 3

For Variant 3, we implement the square network with $q = 15$ layers and each permutation node is initialized by Variant 2. The benchmark is available in Table 3.4. The result shows that we can permute $k = 90000$ messages in around 12 seconds with 46MB communication.

Table 3.4. Performance of the online phase of Variant 3 in three party setting. (k refers to the number of clients. We assume each client sends a 16 byte field element as the message in each execution. Communication is measured by total MB sent per party)

k	Online Time (s)	Online Communication(MB)
10000	0.56	5.12
40000	3.97	20.48
90000	12.68	46.08
160000	27.69	87.92

Performance with More Servers

We take Variant 3 as an example and run it with $k = 10000$ for a different number of servers. The results are available in Table 3.5. We only see a slight increase in online running time when increasing the number of servers. The reason is that our protocols are mostly bottlenecked by the local computation (the local inner product computation), which is independent of the number of servers.

Table 3.5. Online Performance of Variant 3 with more servers ($k = 10000$ for all experiments, the hardware settings are the same as the experiments above.)

n	Online Time (s)	Online Communication(MB)
3	0.56	5.12
5	0.58	10.2
7	0.64	15.3

3.6.3 Offline Phase Benchmark

To illustrate the offline phase are practical, we run the offline phases of all three variants and record their performance in this section. For the first two variants, the offline phase is bottleneck by an k-by-k matrix operation. The result shows that we can run the offline phase of Variant 1 for $k = 1000$ in 3.9 seconds with 32MB communication. With the increase of k, the offline time increases significantly, therefore we recommend the users to use Variant 3 for large k. The offline phase of Variant 2 has almost the same time and communication cost because they are both computational-bounded by the generation of the permutation matrix.

Table 3.6. Performance of the offline phase of Variant 1 in three party setting. (k refers to the number of clients. Communication is measured by total MB sent per party)

k	Offline Time (s)	Online Communication(MB)
1000	3.99	32
3000	120	288
5000	598	800
7000	1804	1568
10000	5731	3200

For Variant 3, the offline phase is responsible to do $q\sqrt{k}$ matrix multiplications, with matrix size $\sqrt{k}$-by-$\sqrt{k}$. We use multi-threading to perform multiple matrix multiplications simultaneously, but each matrix multiplication itself is not optimized by parallelism. we record the offline phase performance in Table 3.7. The benchmark shows that the offline phase of $k = 90000$ only takes about 7 minutes. As for the bottleneck, we observe that at least for k up to 90000, the communication and the computation both take a significant

portion of the time so they are both the bottleneck. We note that the communication cost of this offline phase is large compared with the first two variants.

Table 3.7. Offline phase benchmark for Variant 3. (k is the number of clients, communication is measured by total MB sent per server)

k	Offline Time (s)	Offline Communication(MB)
10000	5.741	480
40000	89.08	3840
90000	448.82	12960

3.6.4 Towards Robustness

Our proposed protocols can support more security properties if built in proper MPC building blocks. As an illustrative example, we show how to achieve a robust online phase for our protocols in a synchronous setting.

Protocol Consturction

As mentioned, the online phase of Variant 2 and Variant 3 only requires secure secret sharing reconstruction as the MPC building block. Therefore, we follow the ideas of [15] and construct a robust Shamir secret-sharing reconstruction using Reed-Solomon decoding [110]. We require $n \geq 3t + 1$ to guarantee that a sufficient number of shares are available as the input of the robust decoding algorithm. Any wrong shares sent by the malicious parties will be corrected by the robust decoding. In a synchronous setting, if malicious parties refuse to send shares, they will be caught by the honest online parties, and honest online parties can use arbitrary shares as malicious parties' shares, and treat them as the "wrong shares".

We implement this idea using MP-SPDZ with the "malicious-shamir-party" back-end. The original back-end achieves secure-with-abort sharing reconstruction in a $n \geq 2t + 1$ setting. The party will use the first $t + 1$ shares to reconstruct a polynomial and use the rest t points to confirm all points correspond to the same polynomial. If any inconsistency occurs, the protocol will abort. To achieve robustness, we change the model to be $n \geq 3t + 1$ and replace its share reconstruction with a robust one. The Reed-Solomon decoding algorithm we choose

is from [110]. With this design, the share reconstruction will have the same performance as the non-robust version if there are no malicious behaviors. In most scenarios, the probability that the malicious behaviors happen is low, therefore this design is beneficial to the overall performance.

Evaluation of the Robust Variant 3

We test the performance of the robust versions of our protocols in the AWS cluster, the hardware and network setting are the same as in the rest of the experiments. The only difference is that we use $n = 4, t = 1$ to fulfill the requirement of robust MPC. In best cases, the performance is the same as the non-robust version. To illustrate the worst case, we conduct an experiment where we trigger the malicious behaviors in every share reconstruction by forcing one party to send 0 all the time as its shares. We build up the robust implementation of Variant 3 by using Variant 2 as the building block, and the benchmark result is available in Table 3.8. In worst cases, the performance is approximately 2× more than the non-robust version.

Table 3.8. Online phase performance of the **robust** Variant 3 in $(n = 4, t = 1)$ setting. For data in this table, We trigger the malicious behavior by always forcing one party to send wrong shares to simulate the worst case.

k	Online Time (s)
10000	1.75
40000	8.62
90000	24.05
160000	48.14

3.6.5 Performance Comparison

We present a comparison to PowerMix [15] and Blinder [61] as they share the closest theoretical complexity with ours.

Compared with our work, PowerMix shares the same online communication complexity and online rounds. Besides, the client computation cost and communication cost are exactly the same between the two protocols. In Powermix, it takes around 140 seconds to mix

$k = 1000$ messages, while our protocol (Variant 2) takes around 0.02 seconds. The main reason that we outperform Powermix is the online computation complexity ($O(k^3)$ vs $O(k^2)$), and we also confirm the online computation is the main bottleneck of the whole protocol.

For Blinder, our protocol and the Blinder protocol share the same online communication complexity and online rounds. The first two variants of our protocol share the same online computation complexity with blinder, while Variant 3 has a better online computation complexity by a factor of $O(\sqrt{k})$. Blinder's benchmark is based on five MPC parties as they require $N \geq 4t + 1$, and the closest test case we can find is $k = 100000$, with message size being 160B. Our case is $k = 100000$ and the message size is 16B, ten times smaller than Blinder's test case. Blinder's non-robust test case takes around 8 minutes to finish in their CPU version and around 40 seconds in GPU version. Our protocol only implements CPU version and it takes around 14 seconds to finish. As mentioned earlier, we can support larger messages by re-running the protocol for all message pieces using the same permutation matrices, so we can simply multiply our performance numbers with 10 for a fair comparison. With the 10× factor incorporated, we outperform the blinder CPU version protocol by around 3.5×, and their GPU version is better than ours. We expect a similar performance gain if our protocol can be implemented in GPU version as most of the local computation can be done in parallel. We will take it as one of the future works. The comparison above is based on the non-robust versions of both works. If we take the robustness into the picture, our protocols outperform blinder by a factor of 1.7× to 3.5×, depending on the frequency of malicious behaviors.

What's more, the main difference between our protocol and blinder protocol is the client cost, the computation and communication cost of our protocol is $O(1)$ while the blinder's client cost is $O(\sqrt{k})$. Therefore, when $k = 100000$, the client cost of our protocol is 316x cheaper than blinder in terms of both the communication cost and the computation cost. This performance difference makes our protocols preferred choices for clients with limited hardware resources.

For Clarion [62], we realize that it outperforms our protocols when it comes to a large volume of inputs (e.g. $k \geq 10^5$). However, Clarion cannot support security properties like robustness. Therefore both works have their own advantages and use cases.

4. Polymath: LOW-LATENCY MPC VIA SECURE POLYNOMIAL EVALUATION

So far, we focus on anonymous communication and design multiple protocols to make anonymous communication efficient, scalable and robust. Starting from this chapter, we will move forward to another prominent application: Privacy-preserving machine learning (PPML). To improve the performance, we design efficient MPC building blocks that are core parts of PPML. These building blocks can also be used in more applications to be widely beneficial. In this chapter, we present Polymath, an MPC toolkit for polynomial operations.

4.1 Motivation and Introduction of Polymath

Typically, in MPC protocols, general-purpose compilers represent any computation/functionality (over private inputs) as a Boolean or an arithmetic circuit. The secure computation proceeds by inductively composing the secure protocols for the atomic elementary Boolean/arithmetic gates [111]–[114]. Although this natural approach is complete, it tends to introduce significant overheads. The privacy-preserving computations are at least a few orders of magnitude slower than their non-private counterparts. Moreover, the overhead increases as we add more parties or consider a more powerful adversary. While the growing demand for privacy, in the form of privacy regulations (for example, GDPR [115] and CPRA/CCPA [116]) and user expectations, is here to drive the use of MPC in a broad spectrum of online applications, the slowdowns among most of the current privacy solutions are far from acceptable.

There are opportunities to significantly improve the efficiency of these secure computation protocols by identifying optimizations within specific computation tasks. However, carefully handcrafting provably-secure MPC solutions for every interesting application is not scalable. Ad hoc protocol design, on the other hand, has historically been riddled with latent security vulnerabilities. Consequently, the time-tested approach is to strike a natural balance between these two extreme techniques by identifying shared algorithmic components underlying several classes of computations of high societal impact. After that, one designs optimized secure

computation solutions for these individual components, thus, permeating the efficiency gains to all application domains relying on them.

This work follows the design principle of identifying and constructing fast and provably secure MPC protocols to evaluate useful high-level algebraic abstractions. We consider secure multi-party polynomial evaluations of scalars and matrices, and present constant-round MPC solutions for them. This class of high-level algebraic primitive finds widespread applications ranging from approximating non-linear functions (e.g. the Sigmoid function) to evaluating decision trees [117].

4.1.1 Our Contribution

We propose Polymath, a versatile, constant-round protocol suite focusing on improving MPC performance of polynomial evaluation over both scalars and matrices. Our protocol suite applies to *any* arithmetic-circuit-based MPC computation, and the numbers of required communication rounds are independent of the number of participating parties, the degree of the evaluated polynomial as well as the matrix dimensions.

Specifically, we focus on the following two functionalities:

(a) Secure Evaluation of Polynomial over Finite Fields

We design 2-round protocols for evaluating multivariate polynomials, and the communication complexity (i.e., the number of communicated bits) only increases linearly with the number of variables and is independent of polynomial degree. Our highly parallelizable protocols employ specially crafted pre-computations and offer significant improvements over the state-of-the-art techniques that require communication rounds logarithmic in the number of variables. We also present 2-round protocols for evaluating univariate polynomials, which extends a secure exponentiation protocol by Damgård et al. [7].

(b) Secure Evaluation of Polynomials over Matrices

We present a 4-round protocol for secure matrix powering/exponentiation and make use of it to evaluate univariate polynomials over matrices securely. Our protocol is not only highly parallelizable but also with a communication complexity independent of the exponent.

Despite the conceptual similarity, our protocols for secure polynomial evaluation for scalars over finite fields and those over matrices are inherently different as the multiplication of finite field elements is commutative, while matrix multiplication is not commutative in general.

Polymath is not only theoretically interesting, but we find it to be practically relevant to any computation that can be represented as a polynomial. We demonstrate this practical relevance using two representative applications.

Application I: Privacy-Preserving Decision Tree Evaluation

We present a solution for privacy-preserving decision tree evaluation as an application of Polymath, and it illustrates that we can achieve high-depth decision tree evaluation within a reasonable time. To the best of our knowledge, our solution is the first to support general n-party setting and high-depth tree evaluations simultaneously.

Polymath can be employed with any arithmetic-circuit MPC library such as SPDZ [118], or SCALE-MAMBA [119]. We implement our protocols using the HoneyBadgerMPC library [5] for robust execution in the asynchronous setting. For a depth 8 complete decision tree model, we can evaluate it with 12 seconds under 4-party setting or 13 seconds under 7-party setting.

Application II: Secure Credit Risk Analysis through a Markov Process

We use our matrix powering protocol to solve the evaluation of a Markov process. Furthermore, we show how one can use this computation to perform credit risk analysis in financial domains. The benchmark shows that in the 4-party setting, we can evaluate the power of 10×10 transition matrices (with the exponent being 1024) in (roughly) half a

second. Furthermore, we can evaluate the power of 1024 for 320×320 transition matrices in around 20 seconds.

4.2 Problem Setting

We consider a standard MPC setting with a set of parties $P_1, P_2, \ldots P_n$ for $n \geq 2$, where the parties are connected via authenticated and secure point-to-point channels, where everyone can send messages to each other at the same time. In this setting, MPC has three distinct phases: parties share their input in phase 1, they participate in multi-party computation over the shared input in phase 2, and finally reconstruct output by combining their shares in phase 3.

The proposed Polymath techniques work across different communication settings and, thus, we do not make any assumptions regarding the bounded-synchrony, partial-synchrony, or asynchrony of the underlying MPC setting; however, we measure our protocols' efficiency in terms of the number of rounds (or the round complexity) and we specify it below for different communication settings.

MPC protocols for the bounded-synchronous communication setting assume that parties proceed in rounds such that the messages sent by any honest party in any given round are delivered to every recipient in the same round. All parties here are assumed to be somewhat synchronized and to be in the same round at all times; thus, the number of rounds is evident from the protocol steps.

For the definition of rounds in the partially-synchronous and asynchronous settings, we follow [120]. Intuitively, when two messages m and m' sent by party P_{i} in an asynchronous MPC are considered to be sent in rounds r and r', $r' > r$, if m' can be computed only after P_{i} has sent m. Here, the number of asynchronous protocol rounds is the maximum (over all honest parties) number of rounds that an honest party uses in the protocol execution.

Polymath is also agnostic to the underlying adversary model, and we leave the adversary model as well as communication setting discussion to the individual application setting.

4.3 Secure Computations of Polynomial Evaluation

4.3.1 Secure Computation for Univariate Polynomials

Univariate polynomials can be written in a standard form: $P(x) = \sum_{i=0}^{n} a_i x^i \in F_p[x]$ where $a_i \in F_p$ are plaintext coefficients. As a result, the problem of evaluating univariate polynomials reduces to computing the power of the secret share $[x]$.

With the protocol proposed by Damgård et al. [7] we can compute all required $[x^i]$ in parallel, then multiply them with corresponding plaintext coefficients. The whole process takes two rounds since the second part is local computation. See Algorithm 6.

Algorithm 6: Univariate Polynomial Evaluation based on [7]

Input : $[x]$, a polynomial $P(x) = \sum_{i=0}^{n} a_i x^i$
Output : $P([x])$
Pre-computation : $[r_1], \ldots, [r_n]$ and $[r_1^{-1}], \ldots, [r_n^{-n}]$ where r_is are random values.

1 **for** i $\leftarrow 1$ *to* n **do**
2 $\quad [x^i] = \textbf{Pow}([x], i)$ `// Round 1 & 2`
3 **for** i $\leftarrow 0$ *to* n **do**
4 $\quad P_i([x^i]) = [x^i] \cdot a_i$
5 **return** $P([x]) = \sum P_i([x^i])$

4.3.2 How to calculate multi-variable polynomials

We start with some basic protocols to compute simple multi-variable terms, then we extend the protocol step-by-step. Finally, we show how we achieve the evaluation of high-degree multi-variable polynomials.

Efficient Way to Calculate $[xyz]$

Beaver triple technique illustrates how to multiply 2 variables with pre-computed triples. We extend this idea and explore how to calculate multiplication of 3 variables $[x]$, $[y]$, and $[z]$ in one round. The precomputation required by our protocol is the following: $[a]$, $[b]$, $[c]$, and $[abc]$ where a, b, c are random elements. Similar to Beaver's idea, $[a]$, $[b]$, and $[c]$ are used to blind $[x]$, $[y]$, and $[z]$. The computation is based on the following formula:

$$[(x-a)(y-b)(z-c)] = [xyz] - [xyc] - [xbz] + [xbc]$$
$$-[ayz] + [ayc] + [abz] + [abc]$$

We first open the values $(x-a), (y-b)$, and $(z-c)$. Then we can get the following formula by taking the opened values as constants and combining terms with the same prefix:

$$(x-a)(y-b)(z-c) = [xyz] - [xc](y-b)$$
$$-[bz](x-a) - [ay](z-c) - [abc]$$

Algorithm 7: Tri-variate Term Evaluation

Input : $[x], [y], [z]$
Output : $[xyz]$
Pre-computation : $[a], [b], [c]$ and $[abc]$ where a, b, c are random values. Three Beaver triples.

1 $[x-a] = [x] - [a]$
2 $[y-b] = [y] - [b]$
3 $[z-c] = [z] - [c]$
4 $(x-a) = \textbf{Open}([x-a])$ `// Round 1`
5 $(y-b) = \textbf{Open}([y-b])$ `// Round 1`
6 $(z-c) = \textbf{Open}([z-c])$ `// Round 1`
7 $[xc] = \textbf{Mul}([x],[c])$ `// Round 1`
8 $[bz] = \textbf{Mul}([b],[z])$ `// Round 1`
9 $[ay] = \textbf{Mul}([a],[y])$ `// Round 1`
10 **return** $(x-a)(y-b)(z-c) + [xc](y-b) + [bz](x-a) + [ay](z-c) - [abc]$

This formula illustrates that the multiplication of three variables reduces to three multiplication of two variables by the help of $[a]$, $[b]$, $[c]$, and $[abc]$. Three normal Beaver triples are required for solving 3 two-variable multiplications.

The protocol is formally described in Algorithm 7. The round complexity is 1 since the opening of $(x-a), (y-b)$, and $(z-c)$ could be done simultaneously with two-variable multiplications. The required invocation of broadcasts is 9 where 3 broadcasts are used to open $(x-a), (y-b)$, and $(z-c)$ and 6 broadcasts deal with two-variable multiplications. As a trade-off, the computation overhead is higher than simply using Beaver multiplications twice. So our method provides an alternative that achieves better performance in high-latency networks.

The proposed method implies that the idea of Beaver multiplication could be extended to more variables. However, the computation and the offline cost increase exponentially with

the number of variables. As a result, we need a better technique to deal with the general multi-variable multiplication. Although we did not include this protocol in our final solution, it could potentially be useful and the same idea could be extended to other fields (e.g. it can be naturally extended to matrix multiplication).

Efficient Way to Calculate Multi-variable Multiplication

Here we take one step further and introduce a method to solve multi-variable multiplication. Suppose we want to calculate the product of $[x_1]$, $[x_2]$, ..., $[x_n]$, our protocol requires pre-computed values in the form of $[a_1]$, $[a_2]$, ..., $[a_n]$, $[(a_1 a_2 \dots a_n)^{-1}]$ where $a_1, a_2, \dots, a_n$ are random values.

The protocol proceeds as follows: First, we use Beaver multiplications to multiply $[x_\mathrm{i}]$ and $[a_\mathrm{i}]$. Then we open all the multiplication results $x_\mathrm{i} a_\mathrm{i}$. Finally, we get the result by multiplying all the opened values together with pre-computed value as follows: $x_1 a_1 x_2 a_2 \dots x_n a_n [(a_1 a_2 \dots a_n)^{-1}]$.

The round complexity of the protocol is 2. And the communication complexity increases linearly with the number of variables. Compared with the method in the previous section, this protocol has advantages both in the round complexity and offline costs. The protocol is formally described in Algorithm 8.

Algorithm 8: Evaluation of degree-1 arbitrary-variate term

Input : $[x_1], [x_2], \dots, [x_n]$
Output : $[x_1 x_2 \dots x_n]$
Pre-computation: $[r_1], [r_2], \dots, [r_n]$ and $[(r_1 r_2 \dots r_n)^{-1}]$ where r_is are random values. n Beaver triples.

1 **for** i $\leftarrow 1$ *to* n **do**
2 $[x_\mathrm{i} r_\mathrm{i}] = \mathbf{Mul}([x_\mathrm{i}], [r_\mathrm{i}])$ `// Round 1`
3 **for** i $\leftarrow 1$ *to* n **do**
4 $x_\mathrm{i} r_\mathrm{i} = \mathbf{Open}([x_\mathrm{i} r_\mathrm{i}])$ `// Round 2`
5 **return** $x_1 r_1 x_2 r_2 \dots x_n r_n \cdot [(r_1 r_2 \dots r_n)^{-1}]$

Efficient Way to Calculate $x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}$

To solve the problem of polynomial evaluation, we need to deal with terms with the most generalized form: $x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}$. We extend the ideas in previous sections to solve this problem. In our protocol we require the pre-computed values in the form of $[r_1]$, $[r_2]$, ..., $[r_n]$, and $[(r_1^{e_1} r_2^{e_2} \dots r_n^{e_n})^{-1}]$. The protocol proceeds as follows:

First we multiply $[x_i]$ with $[r_i]$ for i from 1 to n. Then we open all these products to get all $x_i r_i$. After that, the result could be written as $(x_1 r_1)^{e_1} \dots (x_n r_n)^{e_n} [(r_1^{e_1} r_2^{e_2} \dots r_n^{e_n})^{-1}]$.

The protocol is formally described in Algorithm 9. The round complexity is always 2 and the communication complexity is linear with the number of variables. Given a polynomial, we can apply this protocol to each term in parallel and as a result, any polynomial with arbitrary degrees can be evaluated in two rounds.

Algorithm 9: Evaluation of multi-variate polynomial term

Input : $[x_1], [x_2] \dots [x_n]$ and $e_1, e_2, \dots, e_n$
Output : $[x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}]$
Pre-computation : $[r_1], [r_2], \dots [r_n]$ and $[(r_1^{-e_1} r_2^{-e_2} \dots r_n^{-e_n}]$ where r_is are random values. n Beaver triples..

1 **for** i $\leftarrow$ 1 *to* n **do**
2 $\quad [x_i r_i] =$ **Mul**($[x_i]$,$[r_i]$) `// Round 1`
3 **for** i $\leftarrow$ 1 *to* n **do**
4 $\quad x_i r_i =$ **Open**($[x_i r_i]$) `// Round 2`
5 **for** i $\leftarrow$ 1 *to* n **do**
6 $\quad y_i = (x_i r_i)^{e_i}$ `// local computation`
7 **return** $y_1 y_2 \dots y_n \cdot [(r_1^{-e_1} r_2^{-e_2} \dots r_n^{-e_n}]$

4.3.3 Offline Phase

For each polynomial term $x_1^{e_1} x_2^{e_2} \dots x_n^{e_n}$, the required pre-computed values are $[r_1], \dots, [r_n]$, and $[r_1^{-e_1} r_2^{-e_2} \dots r_n^{-e_n}]$. The straightforward way to generate these values is to calculate powers first and then multiply all random terms together. Given a random share $[r]$, it requires approximately $log(e)$ multiplications to compute $[r^e]$ by running Beaver multiplication repeatedly.

As a result, the offline phase of Algorithm 9 requires the generation of $2n$ random elements, $n + \sum log(\mathrm{e_i})$ MPC multiplication and n reconstruction, where n is the number of variables. The communication complexity of each of the above operations could be linear with the number of parties if we pick proper sub-protocols (e.g. [23]). The overall round complexity is $O(log(max(\mathrm{e_i})) + log(n))$ considering both the computation of $[r^{\mathrm{e_i}}]$ and the multiplication of all $[r_{\mathrm{i}}^{\mathrm{e_i}}]$ can be parallelized. Unlike the online phase, the complexity of the offline phase depends on the degrees of the polynomial.

4.4 Secure Computation of Polynomials of Matrices

In this section, we discuss polynomial evaluation over matrices where each matrix entry is a finite field element. Although it is conceptually similar to the scalar polynomial evaluation, the protocols for matrices are inherently different because matrix multiplication is not commutative in general.

4.4.1 Multiplying an arbitrary number of matrices

To solve polynomials over matrices, we need to evaluate the terms that consist of the product of multiple matrices. Bar-Ilan and Beaver [121] propose a constant-round protocol to achieve it. To multiply n matrices $X_1, X_2, \ldots . X_n$ (for simplicity of description, we assume all of them are k by k square matrices, while this solution also works for matrices with arbitrary dimensions), it requires $n + 1$ random matrices $R_0, R_1, R_2, \ldots . R_n$, which have the same dimensions as X_{i}s and contain random values, and their corresponding inverse matrices $R_0^{-1}, R_1^{-1}, R_2^{-1}, \ldots . R_n^{-1}$.

The protocol is illustrated in Algorithm 10. The round complexity of the protocol is four. The first two rounds are used to do multiplications for all X_{i}s and the third round is to open them. The final round is used to multiply $[R_0]$, $R_0^{-1} X_1 X_2 \ldots X_n R_n$, and $[R_n^{-1}]$. The protocol also works well when X_{i}s are not square matrices and have different dimensions. Parties only need to generate the precomputed random matrices R_{i} accordingly so that the dimensions of matrices match each other.

Algorithm 10: Multiply an arbitrary number of matrices in [121]

Input : $[X_1], [X_2], \ldots, [X_n]$
Output : $[X_1 X_2 \ldots X_n]$
Pre-computation : $[R_0], \ldots, [R_n]$ and $[R_0^{-1}], \ldots, [R_n^{-1}]$

for i $\leftarrow 1$ *to* n **do**
  $[R_{\mathrm{i}-1} X_{\mathrm{i}}] = \mathbf{Mul}([R_{\mathrm{i}-1}],[X_{\mathrm{i}}])$
  $[R_{\mathrm{i}-1} X_{\mathrm{i}} R_{\mathrm{i}}^{-1}] = \mathbf{Mul}([R_{\mathrm{i}-1} X_{\mathrm{i}}],[R_{\mathrm{i}}^{-1}])$
for i $\leftarrow 1$ *to* n **do**
  $R_{\mathrm{i}-1} X_{\mathrm{i}} R_{\mathrm{i}}^{-1} = \mathbf{Open}([R_{\mathrm{i}-1} X_{\mathrm{i}} R_{\mathrm{i}}^{-1}])$
$R_0 X_1 \ldots X_n R_n^{-1} = \prod_{\mathrm{i}=1}^{n} R_{\mathrm{i}-1} X_{\mathrm{i}} R_{\mathrm{i}}^{-1}$
return $\mathbf{Mul}([R_0^{-1}] \cdot R_0 X_1 \ldots X_n R_n^{-1}, [R_n])$

The other alternative is to apply Beaver matrix multiplication for $\log n$ rounds. In the first round, we multiply every two matrices together so that the number of resulting matrices is reduced by half. We keep doing it iteratively, and after $\log n$ rounds, there is only one matrix left which is the result. Compared with this idea, the protocol in [121] has constant round complexity but as a trade-off, it requires more local computation and communication.

We implement both of them in HoneybadgerMPC to compare the performance of these two protocols. the implementation detail and the benchmark result are elaborated later in Benchmark Section. The result demonstrates that the second method outperforms the first in almost all test cases. The reason is due to the nature of matrix multiplication: the local computation and communication become the bottleneck quickly with the increase of the dimensions. And the local computation can easily be measured in terms of seconds, thus canceling the benefits gained by constant round complexity which is usually several hundred milliseconds.

However, we find out that by extending the protocol in [121], we can achieve a very efficient protocol for matrix powering.

4.4.2 Calculating powers of a square matrix

The power of square matrices is a special case of matrix multiplication. In this section, we propose a method to calculate the powers of a square matrix, which saves significant

Algorithm 11: Matrix Powering

Input : matrix in secret shared form with demension k by k: $[X]$ and the exponent e
Output : $[X^{\mathrm{e}}]$
Pre-computation : $[R]$ and $[R^{-1}]$ where R consists of k by k random values

1 $[RX] = \mathbf{Mul}([R], [X])$ `// Round 1`
2 $[Y] = \mathbf{Mul}([RX], [R^{-1}])$ `// Round 2`
3 Y = $\mathbf{Open}([Y])$ `// Round 3`
4 **return** $\mathbf{Mul}([R^{-1}] \cdot Y^{\mathrm{e}}, [R])$ `// Round 4`

precomputation and communication, especially when computing a large power. However, this method only achieves a lower level of privacy since some information about the input matrix is leaked (for example, the rank of the input matrix). We find this method meaningful as in many use cases it is acceptable to leak that information to save significant computation.

The protocol is described in Algorithm 11. The observation is that we only need one random matrix R in the case of matrix powering, and we only need to calculate and open $[RXR^{-1}]$ one time. It means the round complexity and communication complexity are independent of the power that we want to compute.

The state-of-the-art solution for matrix powering that we know so far is to use Beaver matrix multiplication for roughly $\log \mathrm{e}$ times to get $[X^{\mathrm{e}}]$[1]. As a result, it requires $\log(\mathrm{e})$ rounds involving opening $2 \cdot \log(\mathrm{e})$ matrices. Compared with it, our protocol requires 4 rounds and the communication cost is opening 7 matrices. As for the computation complexity. Each Beaver matrix multiplication takes 3 local matrix multiplications and 6 additions. Therefore, our protocol requires $9 + \log(\mathrm{e})$ local matrix multiplications and 18 additions. Meanwhile, the state-of-the-art protocol needs $O(\log(\mathrm{e}))$ number of multiplications.

4.4.3 Offline Phase

For matrix powering protocol, the required precomputation is in the form of A, A^{-1}, namely a matrix with random elements and its inverse matrix. Here we provide an efficient

[1] ↑If e is not a power of two, we can represent e with its binary representation, then apply similar techniques to compute each binary term.

2-round protocol to compute the inverse matrix given a secret shared matrix. The protocol is summarized in Algorithm 12.

Algorithm 12: Precomputation for Matrix Powering

Input : secret shared matrix $[X]$
Output : $[X^{-1}]$
Pre-computation : $[R]$, matrix R share the same dimension with X and contain random values.

1 $[XR] = \textbf{Mul}([X], [R])$ `// Round 1`
2 $XR = \textbf{Open}([XR])$ `// Round 2`
3 $R^{-1}X^{-1} = (XR)^{-1}$
4 **return** $[R]R^{-1}X^{-1}$

The cost of this protocol includes generating k^2 random shares, one matrix multiplication, and one matrix reconstruction, all of which have communication complexity of $O(k^2)$ for k by k matrices. Thus the cost of the offline phase is of the same order of magnitude as the online phase.

The above protocol is efficient but there is a very small probability of failure since XR may not have an inverse. The probability of a random matrix being not invertible is less than $\frac{1}{p} + \frac{1}{p^2}$ where p is the size of the field, and it is small enough for us to use in practice.

4.4.4 Security Against Malicious Adversary

We claim our protocols' security in both semi-honest and malicious settings and elaborate our claim here. Our protocols utilize other protocols as building blocks (such as secret sharing schemes). Beyond that, our protocol does not reveal any additional information (with the exception of the protocol Matrix Powering, which reveals nothing beyond the conjugacy class). Therefore, to make our protocol secure against malicious adversaries, we only need to make sure we use linear secret sharing-based MPC that is secure against malicious adversaries. In general, the malicious security of our protocols can be derived from the malicious security of the underlying linear secret sharing-based MPC library.

4.5 Application:Privacy Preserving Decision Tree Evaluation

In recent years, privacy-preserving machine learning becomes more and more prominent and many works [114], [117], [122], [123] are deployed to achieve secure model training or secure inferences. In this work, we focus our efforts on decision trees, one of the common classifiers used in many fields such as medical treatment and finance. To the best of our knowledge, our work provides the first solution for privacy-preserving decision tree evaluation that can be applied to a general multi-party setting with the capability of evaluating high-depth models. We implement our protocols and show that our protocol can achieve very high-precision prediction for decision tree models within a reasonable time.

4.5.1 System Model

We consider three kinds of parties in our system model: model holders, service providers, and clients. We assume model holders are parties that hold the trained model and would like to outsource the models to service providers in a secure fashion. Clients have secret input data and expect the prediction value by the trained model. The data flow is as follows in a secret sharing based setting: model holders secret-share their models to service providers, and clients also secret-share the private input to service providers. The service providers run MPC protocols to achieve decision tree evaluation with the private tree model and private user input.

4.5.2 Technical Overview

To give a general view of our protocol, we represent the trained decision tree models as two kinds of nodes: leaf nodes and non-leaf nodes. The non-leaf node contains a comparison operation so that different branches will be chosen based on the comparison. The leaf nodes store the result of predictions and inspired by [117], we leverage polynomials to represent the leaf node where the degree of the polynomial equals the length of the path between the leaf node and the root. In previous works, evaluating high-degree polynomials is treated as a hard problem so most works only focus on decision trees with low depth(e.g. [117] chose the

max depth to be 8). However, with our protocols we are able to solve high-degree polynomial evaluation more efficiently, thus giving us the advantage that we can evaluate higher-depth trees to get better accuracy.

As a result, we need to solve two problems in a privacy-preserving manner: secure comparison and secure polynomial evaluation. For secure comparison, we can use any existing secure comparison protocols. For secure polynomial evaluation, we use our protocols introduced in Section 4.3.

Secure decision tree forest evaluation can also be achieved by combining our solution with secure multi-party aggregation protocols [124].

4.5.3 Decision Tree Representation

Decision tree representation with polynomials

Giacomelli et al. [117] describe how a tree can be represented by polynomials. That representation is a perfect match with our protocols described in Section 4.3 so we choose to use it in our solution. We give a brief introduction here and we refer readers to [117] for more details.

In general, a decision tree T can be represented by two types of nodes: non-leaf nodes and leaf nodes. We can always take T as a binary tree since all trees can be transformed into a binary fashion. Also to hide the topology of the decision tree, we simply transform the decision tree to be a complete tree by adding dummy nodes. The input to the tree is a vector $X = (X[1], X[2], \ldots, X[n])$ where we call $X[\mathrm{i}]$ features and the output is the prediction value y. In each non-leaf node, a comparison operation is defined by a pair $(\mathrm{j_i}, t_\mathrm{i})$ where $\mathrm{j_i}$ ranges from 1 to n and t_i is a threshold value. This means at this node, we will choose the left branch if $X[\mathrm{j_i}] < t_\mathrm{i}$, otherwise we choose the right branch. Given the input x, we follow this rule to traverse the tree starting from the root and finally reach a leaf node where the prediction value y is stored.

Assume the result of comparison in each non-leaf node N_i to be $x_\mathrm{i} = (X[\mathrm{j_i}] < t_\mathrm{i})? - 1 : 1$. Then for each leaf node, we represent it with the product of d terms of the form $(x_\mathrm{i} - 1)$ and $(x_\mathrm{i} + 1)$ as follows: starting from the root node, $(x_\mathrm{i} - 1)$ is chosen if the root-leaf path chooses

the left branch at node N_{i}, otherwise $(x_{\mathrm{i}}+1)$ is chosen. Since we consider a complete binary tree. So each leaf node L_{i} can be represented as a polynomial P_{i} of degree d containing d factors of $(x_{\mathrm{i}}-1)$ or $(x_{\mathrm{i}}+1)$. Now given an input x, it can be observed that there is only one P_{i} that the evaluation of P_{i} is non-zero. And it means the value stored in the leaf-node L_{i} is the prediction y of the input x.

As a result, we can solve the decision tree evaluation with many comparisons and one single polynomial $T(x)=\sum P(\mathrm{i})\cdot inv_{\mathrm{i}}\cdot y_{\mathrm{i}}$ where $P(\mathrm{i})$ is the polynomial representing the leaf node L_{i}, inv_{i} is the inverse value of $P(\mathrm{i})$ when $P(\mathrm{i})$ is non-zero (the value of $P(\mathrm{i})$ is 2^d if there is even number of $(x_{\mathrm{i}}-1)$, or -2^d otherwise), and y_{i} is the prediction stored on the leaf node L_{i}. Given an input x, only one leaf node L_{i} is reached in the end and so that $P(\mathrm{i})\cdot inv_{\mathrm{i}}=1$ and $P(\mathrm{j})\cdot inv_{\mathrm{j}}=0$ for all $\mathrm{j}\neq\mathrm{i}$, therefore $T(x)=y_{\mathrm{i}}$.

Fixed-point numbers representation

Considering the feature space is typically $\mathbb{R}$, we use fixed-point numbers as data format. Therefore we require fixed-point number algebra in a secure computation scenario. For data representation, we follow the standard proposed in [125], [126] and consider a multi-party computation framework based on Shamir secret sharing over a prime field $\mathbb{Z}_q$. The data type that we consider are signed integers and signed fixed-point numbers. We encode a k-bit integer $x\in\mathbb{Z}_{<k>}=\{-2^k<x\leq 2^k-1\}$ to field element x' in $\mathbb{Z}_q$ through a simple mod operation f: $x'=f(x)=x \bmod q$. As a result, for any two integers x and y and the operations $\circledcirc\in\{+,-,\cdot\}$, we have $x\circledcirc y=f^{-1}(f(x)\circledcirc f(y))$. We require $q>2^{k+1}$ since we have 2^k non-negative numbers and 2^k negative numbers in total. For multiplication, we require $q>2^{2(k+1)}$ to avoid multiplication overflow[125], [126].

As for fixed-point numbers, we encode them as follows: for a signed fixed-point number $x\in\mathbb{Q}_{<k,f>}=\{x=\bar{x}\cdot 2^{-f},\bar{x}\in\mathbb{Z}_{<k>}\}$, we represent it using the corresponding integer $\bar{x}=x\cdot 2^f$. Therefore, the addition, subtraction and multiplication of fixed-point numbers could be achieved by directly doing computations over their integer representation. The multiplication requires more attention since a f-bit truncation on the result is needed to

maintain the precision of the multiplication result. We directly use the truncation protocol introduced in [126] to achieve the goal.

4.5.4 Secure Comparison over fixed-point Numbers

All non-leaf nodes in the decision tree are represented as secure comparisons. Therefore, we require a general n-party secure comparison protocol for this task. In our solution, the secure comparison protocol can be treated as a plug-and-play module, and many protocols in the literature [123], [125]–[127] can be applied here.

We choose to implement the secure comparison protocol in [126] to make our decision tree solution complete. This secure comparison protocol is designed for the general n-party setting. It is a good match to our solution as we derive the fix-point number representation from the same work. Besides, it can be applied against both semi-honest and malicious adversaries. As mentioned in [126], security against a malicious adversary can be achieved through using malicious secure building blocks such as verifiable secret sharing(VSS)[2]. We refer readers to [126] for more details of the protocol.

The benchmark illustrates that the secure comparison protocol becomes the bottleneck as our polynomial evaluation protocol is significantly faster. Indeed, the overall performance could improve significantly once a more efficient secure comparison protocol is available in the future. The goal of this work is to provide highly efficient building blocks for secure polynomial evaluation.

4.5.5 Secure Polynomial Evaluation

Recall that we can solve the decision tree evaluation by solving the polynomial $T(x) = \sum P(\mathrm{i}) \cdot inv_{\mathrm{i}} \cdot y_{\mathrm{i}}$. We apply our polynomial evaluation protocol to each term of polynomials and then sum them up to form the final $T(x)$. For a complete binary decision tree with depth d, $P(\mathrm{i}) \cdot inv_{\mathrm{i}} \cdot y_{\mathrm{i}}$ can be treated as a degree $d+1$ polynomial where inv_{i} is a constant coefficient, $P(\mathrm{i})$ contains d variables in secret sharing form as results of comparisons, and y_{i} is the secret shared prediction for the corresponding leaf node. The degrees of all variables

[2]↑In our implementation, such malicious secure building blocks are provided by HoneybadgerMPC [5].

above are 1, so we only need to use the protocol introduced in Algorithm 8 and all the terms can be computed in parallel in two rounds. Note that since the input values are either -1 or 1, they satisfy the non-zero input requirement of Algorithm 8. After that, we locally sum the terms up to get the final $T(x)$.

4.5.6 Prototype Implementation

We developed prototypes for our decision tree evaluation protocols. We choose the Nursery dataset from the UCI Machine Learning Repository [128] as the data source. Nursery dataset has 12960 number of instances, each of which has 8 features. The original dataset consists of only string data so we mapped them to fixed-point numbers. We used sk-learn library to achieve decision tree training. We directly use the trained model and imported it into the MPC codes.

The MPC protocols are implemented in HoneybadgerMPC [5], a secret sharing based MPC framework that supports malicious security. The implementations are written in Python3, parts of the fixed-point algebra codes are from HoneybadgerMPC itself and we extend its code to support the comparison protocol and secure polynomial evaluation protocol. As for the test environment, we deploy our codes in 4 machines with Intel Xeon E5-263040(40 cores) and 384 GB RAM. These machines are connected through LAN and the latency among each other is around 0.1 ms.

Prime q is a 160-bit integer and thus, the field can support the multiplication of 64-bit integers and 32-bit statistical security for fixed-point algebra [125]. While we do not instantiate malicious behaviors in our experiments, our protocol implementations are malicious secure, and the current benchmark includes the verification overhead. We directly use the malicious secure building blocks provided by the HoneybadgerMPC library.

4.5.7 Experimental Analysis

The original trained model has 385 nodes and achieves 99.6% accuracy. Through tuning the parameters, we are able to get models with different depths, and we extend them to be complete trees to hide the decision tree topology. The experiment result is shown in Table 4.1.

In general, it takes around 12 seconds to finish a depth-8 tree evaluation. And it takes around 100 seconds to evaluate a depth-11 tree. As we can see, the performance of our polynomial evaluation protocol takes only around 10% of the whole time, and the secure comparison protocol is the bottleneck. Overall, the performance of the proposed approach can improve significantly when a more efficient comparison protocol is available. As secure comparison protocol is not the main goal of this work, we omit further discovery for it.

Table 4.1. Decision tree evaluation benchmark using Nursery dataset over 4 parties.

Tree Depth	Time for Comparison (s)	Time for Polynomial (s)	Total Time (s)	Total Bandwidth (MB)
8	11.16	1.20	12.46	3.73
9	22.02	2.72	24.99	7.55
10	44.30	5.69	50.53	15.3
11	87.37	12.67	101.32	30.9

To the best of our knowledge, our protocol is the first to support decision tree evaluation in a general n-party setting.

Introducing Higher Network Latency

Since we run the benchmark on 4 machines in a LAN where the network latency is below one millisecond, We launch another test where we change the communication layer of HoneybadgerMPC and add 100ms latency to simulate real-world settings[3]. The benchmark result demonstrates that the total online time increases by around $200 - 300$ms as expected.

Offline Phase Benchmark

We benchmark the offline phase of our protocol using the same environment as the online phase experiment. The result is shown in Table 4.2. For the offline phase of polynomial evaluation, we get benchmark data by directly running the offline phase code. And for the

[3]↑Note that our protocol requires communication of a few MB of data. Nevertheless, the considered MPC-as-a-service setting often runs in networks with sufficient bandwidth (e.g. AWS EC2 [129]) and bandwidth is not counted as the bottleneck of the protocol. As a result, we only study the influence of network latency in the experiments.

Table 4.2. Offline cost of decision tree solution

Tree depth	Time (secure comparison) (s)	Time (polynomial evaluation)	Bandwidth (secure comparison)	Bandwidth (polynomial evaluation)
8	62.43	3.69	16.9	1.53
9	124.9	7.09	33.8	3.42
10	249.7	14.93	67.6	7.57
11	499.5	31.91	135.2	16.54

* The experiment is run under a 4-party setting with (4,1) Shamir secret sharing.
* The unit of time is second, and the unit of bandwidth is MB.

offline phase of the secure comparison protocol, we count the number of randomness required by the online protocol, then run HoneybadgerMPC offline phase codes to generate the same amount of randomness, and record the execution time and bandwidth as the offline phase benchmark. The result shows that the cost of the offline phase is acceptable. To give an illustrative example, for the evaluation of a depth-8 decision tree model, the offline phase cost takes around 84% of the total time cost and 83% of the total bandwidth cost.

Analysis with Another Dataset

To test if the performance is sensitive to the training dataset or resulting models. We test our protocols again using the Boston housing dataset, which is the largest dataset measured in the work of Kiss et al. [130].

The result illustrates that the general performance looks almost the same as the nursery dataset. The reason is that the performance is theoretically independent of the training dataset. Instead, it depends only on the depth of the tree. For a model with depth d, the number of comparisons required is always $2^d - 1$ (the internal nodes) and the number of polynomial evaluation calls is 2^d (leaf nodes).

4.6 Application: Secure Markov Process Evaluation

In this section, we demonstrate how the matrix powering protocol can be used to solve secure Markov process evaluation and we provide a concrete example: credit risk analysis.

Table 4.3. Decision tree evaluation benchmark using Nursery dataset over 7 parties with (7, 2) secret sharing.

Tree Depth	Time for Comparison (s)	Time for Polynomial (s)	Total Time (s)	Total Bandwidth (MB)
8	11.75	1.30	13.16	4.98
9	23.71	2.81	26.76	10.08
10	45.41	6.17	52.11	20.42
11	89.94	13.32	104.51	41.3

4.6.1 Markov Chain Introduction

Markov chain is a common-used model to describe a process with multiple states where the future state only depends on the current state and not on the past. It is used in many applications to model randomness, ranging from biology to economics. We include all possible states into a state space $\mathbf{S} = \{s_1, s_2, \ldots, s_k\}$. We say a process satisfies Markov property if the probability P_{ij} of state transitions from s_{i} to s_{j} only depends only on the current state s_{i}. We denote x_{i} to be the state of the process in the time point i, then the Markov property can be described by the following equation:

$P(x_{t+1}$`x_1, x_2, \ldots x_{t}) = P(x_{t+1}`$x_t)$

In this work, we focus on the first-order stationary Markov processes where all probabilities P_{ij} are constant numbers and do not change with time. As a result, we can represent the probabilities with a k-by-k square matrix P, then we can compute $x_{\mathrm{i}} = x_0 \cdot P^{\mathrm{i}}$. This is the equation that we need to solve for the Markov process evaluation at the time point i.

4.6.2 The Markov Chain Application: Credit Risk Analysis

Credit risk analysis is a significant task for banks since the profits and risks of banks are directly linked with the credit quality of their customers, and the credit quality of customers is modeled through the Markov process in many economy works [131], [132]. In these works, a transition matrix is used to explain the transition of creditor quality, and many methods are introduced to generate proper transition matrices given the data of creditors. The transition matrix is valuable in the sense that it can be used to predict the credit quality migration for

new customer who shares similar financial backgrounds. However, there is almost no research on privacy of credit risk analysis.

We observe that privacy can fit the problem of credit risk analysis quite well due to the following reasons: transition matrices are valuable assets since it is trained on sensitive data of creditors and can be used for predictions of new customers. As a result, it is a perfect match that multi-party computation can be applied here so that one bank can train transition matrices, sell matrices to other banks by providing prediction services, and meanwhile keep the transition matrices hidden. And this pattern can be applied to many Markov chain models in fields beyond finance.

4.6.3 System Model

We abstract the scenarios into an MPC-as-a-service model. There are three types of parties: model holders who own the transition matrices, servers who collaborate to provide MPC services, and clients who want to get the prediction value given the Markov model. Both the transition matrices and the input of clients are considered private data and should keep hidden from other parties. The data flow is as follows: the model holders (e.g. banks) outsource their transition matrices to the servers through secret sharing, the clients also secret-share their private input to servers so that servers can run MPC protocols to finish the evaluation. We assume servers are fully connected with each other and keep online to provide MPC service. Model holders only need to upload their models to servers then they are not required to be online.

The adversary model is the same as our general setting: we support both semi-honest adversary and malicious adversary by picking different sub-protocols such as Shamir secret sharing and verifiable secret sharing.

4.6.4 Secure Evaluation of Markov Process

As mentioned, the equation that we need to solve is as follows: $x_{\mathrm{i}} = x_0 \cdot P^{\mathrm{i}}$ where x_0 is the private input from clients in the form of a vector, and P is the trained transition matrix. The core part of the solution is how to efficiently compute matrix powers and we can use our

protocol to achieve it. Recall that we can finish the matrix power computation in four rounds with less communication and computation compared to previous works. As a result, we can achieve the evaluation for the Markov process in five rounds, where the first four rounds are used to compute the power of the transition matrix and the last round is used to multiply P^{i} and x_0.

4.6.5 Experiments for Evaluation of Markov Process

We build up the prototype using HoneybadgerMPC [5] as the MPC framework. The implementation of matrix multiplication and powering are implemented with both Python and C++. The Python codes are mainly responsible for the operations where communication is required. Batch structures are also applied to guarantee the correct round complexity of the protocol. Due to the heavy local computation included in the protocol, mainly matrix multiplications, We implemented C++ code using NTL library to implement them and used file IO as the connection between Python code and C++ code. This file IO caused some overhead that is only because we used a Python framework for MPC, and this overhead can be eliminated when a C/C++ framework is used, which means the performance could have been better. The test environment is 4 cluster machines, each with Intel Xeon E5-263040 (40 cores) cores and 384GB RAM. The latency between these machines is around 0.1ms. The benchmark result for matrix powering is demonstrated in Table 4.5. In the use case of credit risk analysis, the transfer matrix is often small, (e.g. a 9×9 matrix is used in [132]) and our benchmark shows that we can solve 10×10 matrix powering within one second. Besides, we observe that the bottleneck of the protocol is the communication time, and the communication time is independent of the power. Thus our performance almost remains unchanged when we compute higher powers. This observation remains true for small matrices.

To make a comparison, we also implement the typical protocol to compute matrix powering where Beaver matrix multiplication is applied $\log p$ times. The benchmark of the typical protocol is done using the same hardware and it turns out that our protocol outperforms the typical protocol by around 3x when e is small (less than 1024). When e is larger our protocol outperforms the typical protocols more as our communication is independent of the power.

Table 4.4. Offline cost of Matrix powering

Matrix dimension	Time for random generation (s)	Time for actual offline phase (s)	Total (s)	Bandwidth (MB)
$10 * 10$	0.21	0.53	0.74	0.071
$40 * 40$	3.40	1.51	4.91	1.29
$80 * 80$	12.79	5.78	18.57	5.28

* The experiment is run under a 4-party setting with (4,1) Shamir secret sharing.

Offline phase cost

We benchmark the offline phase of our protocol and the result is shown in Table 4.4. The result shows that the cost of the offline phase is acceptable.

Table 4.5. Benchmark result for proposed Matrix Powering protocol under 4 parties

Dimension	Power	Online time(s)	Bandwidth
10×10	16	0.12	0.04
10×10	1024	0.147	0.04
40×40	16	0.465	0.9
40×40	1024	0.532	0.9
320×320	16	18.552	64.1
320×320	1024	19.367	64.1
320×320	8192	20.848	64.1

* The experiment is run under a 4-party setting with (4,1) Shamir secret sharing.
* Bandwidth is measured through total megabytes(MB) sent per party.

4.6.6 Experiments for Multiplying an Arbitrary Number of Matrices

As mentioned in Section 4.4.1, we implement both protocols for multiplying an arbitrary number of matrices. The test environment is the same as the experiment for Markov process. The protocols are instantiated on 4 parties with (4,1) Shamir secret sharing. The detailed benchmark result is shown in Table 4.6. We observe that Beaver matrix multiplication

Table 4.6. Benchmark comparison for multiplying multiple matrices between the method of Bar-Ilan and Beaver [121] and Beaver Matrix Multiplication. The experiments is under 4-party setting. k represents the number of matrices to be multiplied. Bandwidth is measured through total megabytes sent out per party.

Dimension	k	Bar-Ilan and Beaver [121]			Beaver Matrix Multiplication		
		Computation time(s)	Communication time(s)	Bandwidth (MB)	Computation time(s)	Communication time(s)	Bandwidth (MB)
10×10	16	0.07s	0.3	0.5	0.03	0.173	0.17
10×10	64	0.16s	0.86	2.1	0.12	0.71	0.7
20×20	16	0.2s	1.1	2.32	0.07	0.51	0.72
20×20	64	0.82s	2.82	9.07	0.31	0.95	2.99
80×80	16	3.23s	9.07	43.3	0.89	3.06	11.67
80×80	64	20.883s	35.71	169.5	5.82	14.65	49.3

outperforms the protocol of Bar-Ilan and Beaver [121] in all test cases, which confirms our analysis in Section 4.4.1.

4.7 Related works and future directions

To the best of our knowledge, our work is the first to achieve efficient high-degree polynomials evaluation with arbitrary numbers of variables. In [133], Ishai and Kushilevitz try to solve the same problem as we do. The method they chose is to represent the high-degree polynomials into multiple low-degree polynomials with randomness, then solve many low-degree polynomials. Regarding polynomial algebra, Mohassel and Franklin [134] work in the setting of directly performing operations on the polynomials, such as polynomial multiplication, division with remainder, polynomial interpolation, polynomial GCD, etc, while our work is primarily focused on evaluating the polynomials. Dachman-Soled et al. [135] work in the setting of evaluating multivariate polynomials where each party holds different variables as private inputs. In our setting, all the variables are shared among all the parties. Cramer and Damgård [136] also focus on operations of linear algebra, and their work aims at solving the computation of determinant, characteristic polynomial, rank, and the solution space of linear systems of equations, which is a different set of operations than this work.

Privacy-preserving decision tree evaluation was firstly proposed in [137]. Kiss et al. [130] systematize the recent privacy-preserving decision tree training/evaluation solutions and in general, these solutions can be divided into two types. The first type relies on homomorphic encryption. For example, Wu et al. [114], Joye and Salehi [138] use additive homomorphic encryption combined with different private comparison protocols to achieve decision tree evaluation. The second type makes use of garbled circuits with efficient private comparison. For example, Tueno et al. [139] represent the decision tree as an array and achieve privacy-preserving evaluation through garbled circuits or oblivious RAM. Due to the nature of homomorphic encryption and garbled circuits, the above protocols focus only on a two-party setting, where the server has private decision tree models and the client holds the private testing input. Another alternative solution is mentioned in [140], which is also based on secret sharing as we do. It requires full-threshold secret sharing and the comparison protocol is designed for 2 parties, therefore this solution is also limited to 2-party setting. Compared with these works, our work is built on secret sharing and inherently supports the general n-party setting.

5. IMPROVED SECURE COMPARISON PROTOCOL USING FUNCTION TABLES

As mentioned in Chapter 4, the main bottleneck of the secure decision tree evaluation becomes the secure comparison protocol. Therefore, in this chapter, we focus on improving the performance of the secure comparison protocol.

5.1 MPC with Pre-computed Function Table

For our secure comparison construction, we follow the standard offline-online model of MPC, where the computation is separated into an (input independent) offline phase and an online phase. In the offline phase, input-independent correlated secret sharings are generated that are consumed during the online phase.

As the communication and round complexity of the online phase are critical for the performance of an MPC protocol, our goal is to optimize the online phase and try to push more cryptographically expensive operations to the offline phase. In particular, we build upon the secure precomputed function (SPF) table approach introduced by Ishai et al. [141], which offers a very efficient online MPC phase for computing any function.

The intuition behind the SPF protocol is to precompute the function output for all possible inputs to generate function tables in the offline phase, which allows the online phase to be simple function lookups (which is highly efficient).

This approach manages the small input domains well; however, the size of the SPF table increases exponentially with the input size, and it is not practical to use for applications using moderate-sized input domains. While our construction uses SPF tables to achieve a highly efficient online phase, as a key novelty, we significantly reduce the function table size.

5.2 Protocol Overview

The goal of the protocol is to compute the $f_{LTZ}(\cdot)$ for a secret-shared $[x] \in \mathbb{Z}_{2^n}$. First, both online parties compute $[x + r]$ and open it, where r is a random mask generated during the offline phase. Second, they divide the binary representation of $x + r$ into k blocks, where

each block has ℓ bits such that $k \cdot \ell = n$.[1]Third, the online parties perform membership testing functions for all the blocks, and get the intermediate outputs, labeled $a_1, a_2, \ldots a_k$. The intermediate outputs can either be Positive(P), Negative(N), or Undetermined(U)[2]. As the membership testing function has small input domains, the offline party can precompute the function tables for it such that the online phase of the block membership testing is extremely fast. Finally, the parties open the intermediate outputs, and use the intermediate outputs as inputs to the recombination function table to produce the final output.

In the offline phase, P_3 generates the precomputation and function tables needed for the protocol, and secret shares them to P_1 and P_2. We have designed multiple sub-functionalities to collaboratively achieve $f_{LTZ}(\cdot)$, and they are explained in detail below.

5.3 An Illustrative Example of Our Approach

In Figure 5.1, we show an concrete example of how we do problem size reduction. It can be clearly seen that there are at most 2 Undetermined rows in the first block, and for all the following blocks, there is at most one Undetermined row. Therefore, we need exactly two function tables for all the blocks except the first block, such that each table is responsible for one "Undetermined".

5.4 Building Blocks

We start by introducing the building blocks used in our protocol. Then we explain how to use these functions to construct a full protocol.

We use superscripts to denote public parameters of functions, while subscripts denote secret parameters that need to be kept hidden from the online parties. The secret parameters set during the offline phase are $r \in \mathbb{Z}_{2^n}$, $r_1, r_2, \ldots, r_k \in F_3$, and $\beta \in \{0, 1\}$. r is used to mask the input x, r_{i} is used to mask the i-th intermediate result, and β is used to mask the selection bit.

[1]↑ If n is not a multiple of k, then one chooses the k block-lengths such that any two block-lengths are either identical or differ by one. For example, if $n = 8$ and $k = 3$ then block-lengths are $3, 3$, and 2. For the simplicity of the presentation, this minor detail is omitted; however, our implementation addresses this subtlety while creating the partitions.

[2]↑The reconstructed intermediate values are randomly masked such that the actual values are kept secret.

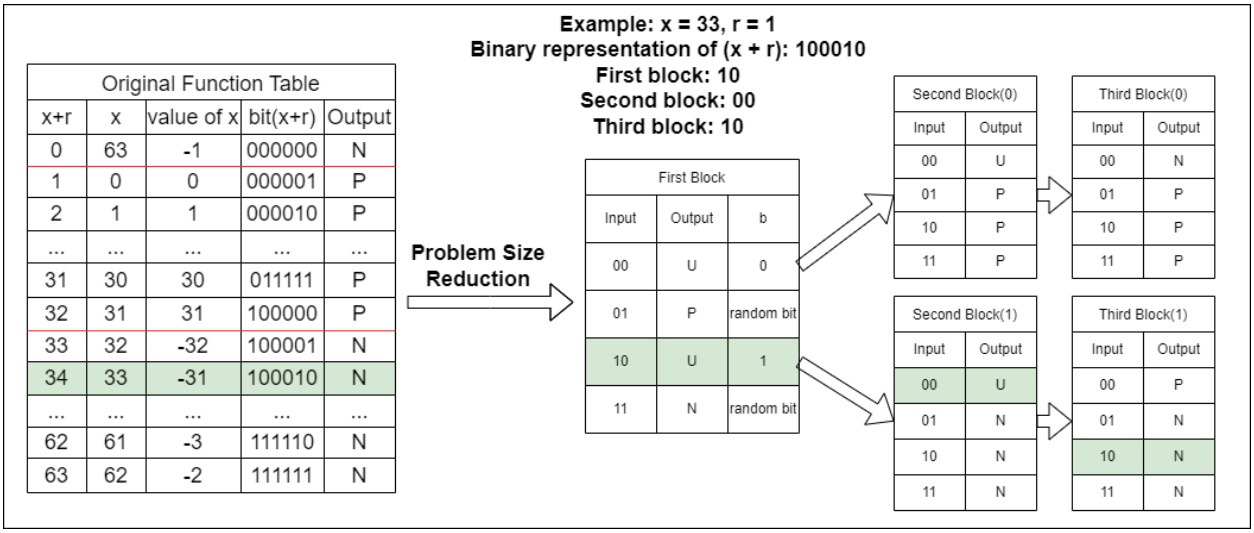

Figure 5.1. An example of problem size reduction where $\lambda = 6, k = 3, \ell = 2$. The left hand side is the overall function table before problem size reduction. The right hand side is the function table structure used in our protocol. "U" means Undetermined, "P" means Positive, and "N" means negative. The red lines in original function tables indicates the gaps when numbers change the sign bit. If any row in function tables includes the gaps, its result is "Undetermined". A significant observation is that there could be at most 2 such gaps, which is the reason why we only need two tables for all the blocks except the first block.

The functions $SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}$ are the coarse-grained membership testing functions that test if a block of input can directly tell us the overall output. $SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}$ takes public parameter $\mathrm{i} \in \{1, 2, \ldots, k\}$, which represents which of the k block this function is for, and it is defined as a family of functions over the secret parameters r, r_{i}, and β.

We start from the first block, the block result is positive if all values that depend on this block (has this block input as prefix) are positive. It is negative if all values that depend on this block are negative, and it is undetermined if some values are positive and some are negative. When the block result of the first block is undetermined, we take the first two blocks as the prefix and check the second block. Since revealing this intermediate result could reveal information on the input, we mask them using the secret parameter r_{i}.

Formally, we define a function $SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}(y_1y_2\dots y_\ell,\widetilde{b}):\{0,1\}^\ell\times\{0,1\}\to F_3$ such that

$$SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}(y_1y_2\dots y_\ell,\widetilde{b})=\begin{cases}0+r_{\mathrm{i}}, & \text{if } z_0\notin N+r, z_1\notin N+r\\ 1+r_{\mathrm{i}}, & \text{if } z_0\in N+r, z_1\in N+r\\ 2+r_{\mathrm{i}}, & \text{otherwise.}\end{cases}$$

where $\vec{z_0}=\underbrace{\mathrm{prefix}_{\mathrm{i},r,\beta\oplus\widetilde{b}}\|y_1y_2\dots y_\ell\|00\dots0}_{n\text{-bits}}$ and $\vec{z_1}=\underbrace{\mathrm{prefix}_{\mathrm{i},r,\beta\oplus\widetilde{b}}\|y_1y_2\dots y_\ell\|11\dots1}_{n\text{-bits}}$, and $\mathrm{prefix}_{\mathrm{i},r,\beta,\widetilde{b}}\in\{0,1\}^{\ell\cdot(\mathrm{i}-1)}$.

Additionally, if $z_0\in N+r$ and $z_1\notin N+r$, then $\mathrm{prefix}_{\mathrm{i}+1,r,0}=\mathrm{prefix}_{\mathrm{i},r,0}\|y_1y_2\dots y_\ell$. If $z_0\notin N+r$ and $z_1\in N+r$, then $\mathrm{prefix}_{\mathrm{i}+1,r,1}=\mathrm{prefix}_{\mathrm{i},r,1}\|y_1y_2\dots y_\ell$. Besides, we use the value 0 in the output above to indicate "Positive", 1 as "Negative", and 2 as "Undetermined".

Next, we explain the function $BLTZ_{r,\beta}$, and the use of the selection bit $\widetilde{b}$ in $SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}(y_1y_2\dots y_\ell,\widetilde{b})$. We note that the result of Undetermined can only appear in at most two rows of the function table of the first block. The reason is that the block result is Undetermined if and only if the numbers represented by that row contain both positive numbers and negative numbers. As the numbers in the function table are consecutive and in ascending order, we only have two such cases: When a block contains both 0 and -1, and when a block contains both $2^{n-1}-1$ and -2^{n-1}.

Since there are at most two uncertain rows in the first block, we build two function tables for the second block (and all the following blocks), one table for each uncertain row in the first block. The input $\widetilde{b}$ of $SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}(y_1y_2\dots y_\ell,\widetilde{b})$ is used to indicate if the first group of the tables are used or the second. The goal of $BLTZ_{r,\beta}$ is to determine $\widetilde{b}$ given the first block of the input, such that b can be available for following $SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}$ executions. We use $\widetilde{b}$ to represent the randomly masked version of b, and $\widetilde{b}$ will be reconstructed during the online phase.

Formally, we define a function $BLTZ_{r,\beta}(y_1y_2\dots y_\ell):\{0,1\}^\ell\to\{0,1\}$ such that

$$BLTZ_{r,\beta}(y_1y_2\dots y_\ell)=\begin{cases}0\oplus\beta, & \text{if } z_0\in N+r, z_1\notin N+r\\ 1\oplus\beta, & \text{otherwise.}\end{cases}$$

Algorithm 13: Less Than Zero $\mathbf{LTZ}([x])$

Input : $[x]$
Output : $[s]$
Pre-computation : $, [r], P_{BLTZ_{r,\beta}}, \left\{P_{SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r,\beta}}\right\}_{\mathrm{i}\in\{1,2,\ldots,k\}}, P_{RECOMB_{r_1,r_2,\ldots r_k,\widetilde{b}}}$

$[y] = [x] + [r]$
$y = \mathbf{Open}([y])$ `// Round 1`
$[\widetilde{b}] = \Pi_{BLTZ_{r,\beta}}(\overrightarrow{y}_{1,\ell})$
$\widetilde{b} = \mathbf{Open}([\widetilde{b}])$ `// Round 2`
for $\mathrm{i} \leftarrow 1$ *to* k **do**
  $[\widetilde{a}_{\mathrm{i}}] = \Pi_{SLTZ^{\mathrm{i}}_{r,\beta,r_{\mathrm{i}}}}\left(\overrightarrow{y}_{(\mathrm{i}-1)\cdot\ell+1\ ,\ \mathrm{i}\cdot\ell}, \widetilde{b}\right)$
$(\widetilde{a}_1, \widetilde{a}_2, \ldots \widetilde{a}_k) = \mathbf{Open}([\widetilde{a}_1], [\widetilde{a}_2], \ldots [\widetilde{a}_k])$ `// Round 3`
$[s] = \Pi_{RECOMB_{r_1,r_2,\ldots r_k}}(\widetilde{a}_1, \widetilde{a}_2, \ldots \widetilde{a}_k)$
return $[s]$

Where $\overrightarrow{z_0} = \underbrace{y_1 y_2 \ldots y_\ell \| 00\ldots0}_{n\text{-bits}}$ and $\overrightarrow{z_1} = \underbrace{y_1 y_2 \ldots y_\ell \| 11\ldots1}_{n\text{-bits}}$.

The $RECOMB_{r_1,r_2,\ldots r_k}$ function takes the intermediate outputs produced by $SLTZ^{\mathrm{i},\widetilde{b}}_{r_{\mathrm{i}},r,\beta}$ and combines them to produce the final output. Intuitively, $RECOMB_{r_1,r_2,\ldots r_k}$ outputs the first intermediate output that is not undetermined.

Formally, function $RECOMB_{r_1,r_2,\ldots r_k}(\widetilde{a}_1, \widetilde{a}_2, \ldots \widetilde{a}_k) : F_3^k \rightarrow \{0,1\}$ offers

$$RECOMB_{r_1,r_2,\ldots r_k}(\widetilde{a}_1, \widetilde{a}_2, \ldots \widetilde{a}_k) = \begin{cases} 0, & \text{if } \exists \mathrm{i} \text{ such that} \\ & \forall \mathrm{j} < \mathrm{i}, (\widetilde{a}_{\mathrm{j}} - r_{\mathrm{j}}) = 2 \\ & \text{and } (\widetilde{a}_{\mathrm{i}} - r_{\mathrm{i}}) = 0 \\ 1, & \text{otherwise.} \end{cases}$$

5.5 Protocol Details

5.5.1 Online Phase

Algorithm 13 describes the online phase of the protocol. During the online phase, our protocol proceeds in three rounds. In the first round, the online parties P_1 and P_2 compute and open $y = x + r$. Next, online parties divide the binary representation of y into k pieces

of ℓ bits each. Parties use the first ℓ bits of y as input to evaluate $\Pi_{BLTZ_{r,\beta}}$ and get the result $[\widetilde{b}]$. In the second round, the parties open $[\widetilde{b}]$. Then the parties use $\widetilde{b}$ and the ith blocks of y as inputs to evaluate $\Pi_{SLTZ^{\mathrm{i}}_{r,\beta,r_{\mathrm{i}}}}$ for all i $\in \{1, 2, \ldots, k\}$ and receive share of the k outputs $\widetilde{a}_1, \widetilde{a}_2, \ldots, \widetilde{a}_k$. In the third round, the parties open all the intermediate results $[\widetilde{a}_{\mathrm{i}}]$. Using them as inputs, the parties run the online phase $\Pi_{RECOMB_{r_1,r_2,\ldots,r_k}}$ to receive secret shares of the final output $[s]$.

5.5.2 Offline Phase

During the offline phase, P_3 generates independent and uniformly random values $r \in \mathbb{Z}_{2^n}$, $\beta \in \{0, 1\}$, and r_{i} for i $\in \{1, 2, \ldots, k\}$. Using those values, P_3 selects the corresponding functions from the family of functions. Specifically, P_3 selects $BLTZ_{r,\beta}$, $SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r}$ for i $\in \{1, 2, \ldots, n\}$, and $RECOMB_{r_1,r_2,\ldots r_k}$. P_3 then acts as our variant of SPF compiler with those functions as inputs, and generates the offline correlated randomness specified by the compiler. P_3 then sends the offline correlated randomness to P_1 and P_2, as well as generate secret shares of r to send to P_1 and P_2. Specifically, the offline correlated randomness are $P_{BLTZ_{r,\beta}}$, $P_{SLTZ^{\mathrm{i}}_{r_{\mathrm{i}},r}}$ for i $\in \{1, 2, \ldots, k\}$, and $P_{RECOMB_{r_1,r_2,\ldots r_k}}$.

5.6 High Level Application: Privacy-preserving Neural Network Training/Inference

To illustrate the performance of our protocols, we develop prototypes for our secure comparison protocols. Additionally, to demonstrate the effect of our protocol in a real-world application, we choose to implement our protocol for the privacy-preserving neural network training/inference application.

The state-of-the-art works in this area are Falcon [8] and SecureNN [142]. We observe that the main bottleneck is the evaluation of the ReLU activation function, where the secure comparison protocol is the core building block. In addition to ReLU, the Division and Maxpool functions have high costs, and the secure comparison is also the main reason. Therefore, we think the secure comparison is one of the main bottlenecks, meaning that the use of our protocol should provide a significant performance improvement. We choose to

implement our protocol in C++ and embed it into the Falcon [8] framework by replacing the derivative of ReLU (Falcon's secure comparison implementation) with our protocol.

Table 5.1. Online Phase Benchmark: privacy-preserving neural network training/inference with our LTZ vs Falcon's LTZ. The experiments are run using AWS t3.2xlarge instances with 130ms ping. Communication is measured by total MB sent per party. parameter setting: $n = 32, k = 4, \ell = 8$.

Network	Mode	Online time(s)		Communication (MB)	
		This work	Falcon	This work	Falcon
Network A	training	6.21s	10.14s	22.61MB	33.98MB
	inference	1.83s	3.16s	0.65MB	1.57MB
Network C	training	22.45s	38.94s	584.21MB	1123.35MB
	inference	11.45s	22.39s	79.392MB	199.949MB

5.6.1 Implementation details

As our protocol has two online parties whereas Falcon has three online parties, we need to properly embed our protocol into the Falcon framework. To begin with, Falcon uses replicated secret sharing. If we only consider the first two parties P_1 and P_2, they actually hold a two-party additive secret sharing of the input. Therefore, we can run our secure comparison on P_1 and P_2 without any other changes. However, we still need to make sure the final output s is a three-party replicated share (e.g. $s = s_1 \oplus s_2 \oplus s_3$, and each party P_{i} holds s_{i} and $s_{\mathrm{i}+1}$). To achieve this, we allow the offline party P_3 to generate s_3 and s_1 in advance and take them as the final shares of P_3. In the offline phase, P_3 sends s_1 to P_1 and s_3 to P_2. In the final recombination table, the shares of s_2 are stored instead of the shares of the final output in our original protocol. Therefore, P_1 and P_2 needs an additional round to reconstruct s_2 using the shares.

We can think of the LTZ as a black box, Falcon provides 3-party replicated secret sharings as the input to LTZ. Inside the box, LTZ operates with 2-party additive secret sharing, and the output of LTZ is a 3-party replicated secret sharing again. The security of this construction is straightforward in a semi-honest setting. We leave the malicious setting construction as one of the future works.

Table 5.2. Offline Phase Benchmark for $n = 32, k = 4, \ell = 8$. (Assuming inputs are n-bit ring elements, and are divided into k blocks with each block being ℓ bits in our protocol.)

Network	Mode	#Comparisons	Execution Time	Communication
Network A	training inference	606720 34048	780s 44s	488 MB 28 MB
Network C	training inference	19968000 1324800	7.16 hr 1688s	16.1 GB 1.1 GB

* The benchmark is executed in AWS clusters with t3.2xlarge instances.

5.6.2 Evaluation Results

To compare our protocol against the secure comparison used by Falcon, we choose to evaluate the performances on Network A and Network C from Falcon. Network A is a 3 layer fully connected network with the ReLU activation function after each layer. Network C is a 4 layer network with 2 convolutional and 2 fully-connected layers. This network also uses both Max Pooling and ReLU.

To begin with, we run the micro-benchmark that only considers the performance of the secure comparison. The micro-benchmark is launched in AWS clusters using three t3.2xlarge instances (8 cores and 32GB RAM). The three instances are located in different regions and have an average round trip time of around 130ms. We call this testing environment the distributed setting" for the rest of the thesis. The experiments show that our LTZ takes 0.008 seconds and 0.163MB data transmission per party to do 128 secure comparison. Meanwhile, falcon's secure comparison takes 0.017 seconds and 0.606MB bandwidth. So we can expect a 2× efficiency improvement on running time and 4× improvements on communication here.

Then we run some neural network tests in the distributed setting. We tested two versions of the codebase: the first one is the original Falcon codebase, and the other is the Falcon codebase only with the LTZ function replaced by our protocol. Therefore all the performance difference is caused by the replacement of the LTZ function. For the neural network training, we run 15 forward-backward pass iterations just to show the performance difference. The benchmark result is available in Table 5.1. In general, the neural network training time with

our LTZ is around 30% more efficient. For the inference, the running time is improved by around 1.4×. The communication of our protocol is also significantly cheaper than Falcon.

We also provide the execution time and communication of the offline phase in Table 5.2. The result shows that the cost is acceptable for neural network use cases.

6. CONCLUSION AND FUTURE WORKS

To conclude, we aim at designing efficient MPC building blocks in these years and the research works are in general divided into two categories: building blocks for secure mixing protocols and building blocks for privacy-preserving data analysis. To construct efficient secure mixing protocols, we propose four approaches, each with different trade-offs. And the corresponding building blocks are of independent interest as they can be widely used in more applications. For privacy-preserving machine learning, we start with the privacy-preserving decision tree evaluation, and construct two efficient building blocks for it: secure polynomial evaluation and secure comparison.

In general, we design high-efficient online phase protocols by trying to push as much computation as possible into the offline phase, meanwhile guaranteeing the offline phase is still practical. By this approach, we succeed in reducing the communication and the round complexity of many state-of-the-art MPC building blocks.

As for future works, there are two directions that I think are promising and meaningful. The first one is the combination of MPC and parallel computing. In our previous works, we try to push as much computation as possible into one single round to improve the round complexity. It means that these computations do not depend on each other, and can be executed in parallel. Therefore, most of our works will see a significant performance gain if combined with parallel computing. Besides, if GPU [143] can be utilized properly, the overall performance of our protocols can also be expected to improve. The second direction is to keep researching the same MPC protocols that we worked on, however, we would like to increase the input size (e.g. what if the input volume is larger, what if we work on larger fields). As we live in an era of big data, scalability will be a challenge sooner or later.

REFERENCES

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *USENIX Security Symposium*, 2004.

[2] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency - choose two," in *IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 108–126.

[3] Y. Sun, A. Edmundson, L. Vanbever, O. Li, J. Rexford, M. Chiang, and P. Mittal, "RAPTOR: routing attacks on privacy in tor," in *USENIX Security Symposium*, 2015, pp. 271–286.

[4] T. T. Blog, *One cell is enough to break tor's anonymity*, https://blog.torproject.org/blog/one-cell-enough, Accessed Nov 2018.

[5] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, New York, NY, USA: Association for Computing Machinery, 2019, pp. 887–903, ISBN: 9781450367479.

[6] D. Lu, A. Yu, A. Kate, and H. Maji, "Polymath: Low-latency mpc via secure polynomial evaluations and its applications," *Proceedings on Privacy Enhancing Technologies*, vol. 2022, no. 1, pp. 396–416, 2022. DOI: doi:10.2478/popets-2022-0020. [Online]. Available: https://doi.org/10.2478/popets-2022-0020.

[7] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *Theory of Cryptography*, S. Halevi and T. Rabin, Eds., 2006, pp. 285–304.

[8] S. Wagh, S. Tople, F. Benhamouda, E. Kushilevitz, P. Mittal, and T. Rabin, "FALCON: honest-majority maliciously secure framework for private deep learning," *CoRR*, vol. abs/2004.02229, 2020. arXiv: 2004.02229. [Online]. Available: https://arxiv.org/abs/2004.02229.

[9] Z. Beerliová-Trubíniová and M. Hirt, "Perfectly-secure mpc with linear communication complexity," in *Theory of Cryptography Conference*, 2008, pp. 213–230.

[10] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *Advances in Cryptology—CRYPTO*, 2007, pp. 572–590.

[11] A. Choudhury, M. Hirt, and A. Patra, "Asynchronous multiparty computation with linear communication complexity," in *DISC*, 2013, pp. 388–402.

[12] A. Choudhury and A. Patra, "An efficient framework for unconditionally secure multiparty computation," *IEEE Transactions on Information Theory*, pp. 428–468, 2017.

[13] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[14] A. Choudhury and A. Patra, "Optimally resilient asynchronous mpc with linear communication complexity," in *ICDCN*, 2015, p. 5.

[15] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, New York, NY, USA: Association for Computing Machinery, 2019, pp. 887–903, ISBN: 9781450367479.

[16] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation," in *IEEE Symposium on Security and Privacy (SP)*, 2019.

[17] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure mpc for dishonest majority–or: Breaking the spdz limits," in *European Symposium on Research in Computer Security*, 2013, pp. 1–18.

[18] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making spdz great again," in *Adanvance in Cryptology—EUROCRYPT*, 2018, pp. 158–189.

[19] A. Aly, M. Keller, E. Orsini, D. Rotaru, P. Scholl, N. Smart, and T. Wood, "Scale–mamba v1. 3: Documentation," Tech. Rep., 2019.

[20] M. Keller, E. Orsini, and P. Scholl, "Mascot: Faster malicious arithmetic secure computation with oblivious transfer," in *ACM CCS*, 2016, pp. 830–842.

[21] X. Wang, S. Ranellucci, and J. Katz, "Global-scale secure multiparty computation," in *ACM CCS*, 2017, pp. 39–56.

[22] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *European Symposium on Research in Computer Security*, 2008, pp. 192–206.

[23] A. Barak, M. Hirt, L. Koskas, and Y. Lindell, "An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants," in *ACM CCS*, 2018, pp. 695–712.

[24] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority mpc for malicious adversaries," in *Advances in Cryptology – Crypto*, 2018, pp. 34–64.

[25] N. P. Smart and T. Wood, "Error detection in monotone span programs with application to communication-efficient multi-party computation," in *CT-RSA*, 2019, pp. 210–229.

[26] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, *et al.*, "Secure multiparty computation goes live," in *International Conference on Financial Cryptography and Data Security*, 2009, pp. 325–343.

[27] D. Bogdanov, R. Talviste, and J. Willemson, "Deploying secure multi-party computation for financial data analysis," in *International Conference on Financial Cryptography and Data Security*, 2012, pp. 57–64.

[28] F. Massacci, C. N. Ngo, J. Nie, D. Venturi, and J. Williams, "Futuresmex: Secure, distributed futures market exchange," in *IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 335–353.

[29] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias, "Mcmix: Anonymous messaging via secure multiparty computation," in *USENIX Security Symposium*, 2017, pp. 1217–1234.

[30] F. Eigner, M. Maffei, I. Pryvalov, F. Pampaloni, and A. Kate, "Differentially private data aggregation with optimal utility," in *ACSAC*, 2014, pp. 316–325.

[31] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, "Atom: Horizontally scaling strong anonymity," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 406–422.

[32] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, "Stadium: A distributed metadata-private messaging system," in *26th SOSP*, 2017, pp. 423–440.

[33] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[34] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems.," in *USENIX annual technical conference*, vol. 8, 2010.

[35] M. Castro, B. Liskov, *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[36] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, 2016.

[37] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *ACM EuroSys*, 2018.

[38] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, *Asynchronous multiparty computation: Theory and implementation*, Cryptology ePrint Archive, https://eprint.iacr.org/2008/415, 2008.

[39] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in *ACM STOC*, 1988, pp. 1–10.

[40] Y. Zhang, A. Steele, and M. Blanton, "Picco: A general-purpose compiler for private distributed computation," in *ACM CCS*, 2013, pp. 813–826.

[41] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, *et al.*, "Fairplay-secure two-party computation system.," in *USENIX Security Symposium*, 2004.

[42] A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers, "Fairness in an unfair world: Fair multiparty computation from public bulletin boards," in *ACM CCS*, 2017, pp. 719–728.

[43] G. Asharov and Y. Lindell, "A full proof of the bgw protocol for perfectly secure multiparty computation," *Journal of Cryptology*, pp. 58–151, 2017.

[44] M. Hirt and J. B. Nielsen, "Robust multiparty computation with linear communication complexity," in *Advances in Cryptology—CRYPTO*, 2006, pp. 463–482.

[45] I. Damgård, Y. Ishai, and M. Krøigaard, "Perfectly secure multiparty computation and the computational overhead of cryptography," in *Advances in Cryptology—EUROCRYPT*, 2010, pp. 445–465.

[46] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith, "Scalable multiparty computation with nearly optimal work and resilience," in *Annual International Cryptology Conference*, 2008, pp. 241–261.

[47] A. Choudhury, E. Orsini, A. Patra, and N. P. Smart, "Linear overhead optimally-resilient robust mpc using preprocessing," in *SCN*, 2016, pp. 147–168.

[48] M. Abe and F. Hoshino, "Remarks on mix-network based on permutation networks," in *International Workshop on Public Key Cryptography (PKC)*, 2001, pp. 317–324.

[49] U. Parampalli, K. Ramchen, and V. Teague, "Efficiently shuffling in public," in *International Workshop on Public Key Cryptography*, 2012, pp. 431–448.

[50] A. Czumaj and B. Vöcking, "Thorp shuffling, butterflies, and non-markovian couplings," in *International Colloquium on Automata, Languages, and Programming*, 2014, pp. 344–355.

[51] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "P2P mixing and unlinkable bitcoin transactions," in *NDSS*, 2017.

[52] J. D. Lipson, "Newton's method: A great algebraic algorithm," in *3rd ACM Symposium on Symbolic and Algebraic Computation*, 1976, pp. 260–270.

[53] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft, "Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation," in *Theory of Cryptography Conference*, 2006, pp. 285–304.

[54] A. Bostan, L. Gonzalez-Vega, H. Perdry, and E. Schost, *Complexity issues on newton sums of polynomials*.

[55] A. Kwon, D. Lazar, S. Devadas, and B. Ford, "Riffle," *Privacy Enhancing Technologies*, pp. 115–134, 2016.

[56] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis, "The loopix anonymity system," in *USENIX Security Symposium*, 2017, pp. 16–18.

[57] D. Lazar, Y. Gilad, and N. Zeldovich, "Karaoke: Distributed private messaging immune to passive traffic analysis," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, 2018, pp. 711–725.

[58] H. Corrigan-Gibbs and B. Ford, "Dissent: Accountable anonymous group messaging," in *ACM CCS*, 2010, pp. 340–350.

[59] N. Borisov, G. Danezis, P. Mittal, and P. Tabriz, "Denial of service or denial of security?" In *ACM CCS*, 2007, pp. 92–102.

[60] D. I. Wolinsky, E. Syta, and B. Ford, "Hang with your buddies to resist intersection attacks," in *ACM CCS*, 2013, pp. 1153–1166.

[61] I. Abraham, B. Pinkas, and A. Yanai, "Blinder–scalable, robust anonymous committed broadcast," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1233–1252.

[62] S. Eskandarian and D. Boneh, "Clarion: Anonymous communication from multiparty shuffling protocols," *Cryptology ePrint Archive*, 2021.

[63] V. Shoup *et al.*, *Ntl, a library for doing number theory*, 2005.

[64] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *Proceedings of the 13th USENIX Security Symposium*, 2004, p. 21.

[65] R. Dingledine and N. Mathewson, *Tor Protocol Specification*, https://gitweb.torproject.org/torspec.git?a=blob_plain;hb=HEAD;f=tor-spec.txt, Accessed Feb 2022.

[66] A. Johnson, C. Wacek, R. Jansen, M. Sherr, and P. Syverson, "Users get routed: Traffic correlation on tor by realistic adversaries," in *Proc. ACM SIGSAC conference on Computer & communications security 2013*, 2013, pp. 337–348.

[67] N. S. Evans, R. Dingledine, and C. Grothoff, "A Practical Congestion Attack on Tor Using Long Paths," 2009, pp. 33–50.

[68] K. S. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. C. Sicker, "Low-resource routing attacks against tor," 2007, pp. 11–20.

[69] N. Alexopoulos, A. Kiayias, R. Talviste, and T. Zacharias, "MCMix: Anonymous messaging via secure multiparty computation," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1217–1234, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/alexopoulos.

[70] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: Scalable private messaging resistant to traffic analysis," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 137–152.

[71] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, "Stadium: A distributed metadata-private messaging system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 423–440.

[72] M. Ciampi, M. Ishaq, M. Magdon-Ismail, R. Ostrovsky, and V. Zikas, "Fairmm: A fast and frontrunning-resistant crypto market-maker," *Cryptology ePrint Archive*, 2021.

[73] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges," *arXiv preprint arXiv:1904.05234*, 2019.

[74] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure mpc for dishonest majority–or: Breaking the spdz limits," in *European Symposium on Research in Computer Security*, Springer, 2013, pp. 1–18.

[75] Y. Ishai, E. Kushilevitz, S. Meldgaard, C. Orlandi, and A. Paskin-Cherniavsky, "On the power of correlated randomness in secure computation," in *Theory of Cryptography Conference*, Springer, 2013, pp. 600–620.

[76] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1575–1590.

[77] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency-choose two," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 108–126.

[78] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[79] S. Le Blond, D. Choffnes, W. Zhou, P. Druschel, H. Ballani, and P. Francis, "Towards Efficient Traffic-analysis Resistant Anonymity Networks," in *Proc. ACM SIGCOMM 2013*, 2013, pp. 303–314.

[80] D. Chaum, D. Das, F. Javani, A. Kate, A. Krasnova, J. de Ruiter, and A. T. Sherman, "Cmix: Mixing with minimal real-time asymmetric cryptographic operations," in *ACNS*, 2017.

[81] G. Danezis, R. Dingledine, and N. Mathewson, "Mixminion: Design of a type iii anonymous remailer protocol," in *2003 Symposium on Security and Privacy, 2003.*, 2003, pp. 2–15.

[82] S. Le Blond, D. Choffnes, W. Caldwell, P. Druschel, and N. Merritt, "Herd: A Scalable, Traffic Analysis Resistant Anonymity Network for VoIP Systems," in *Proc. ACM SIGCOMM 2015*, 2015, pp. 639–652.

[83] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of cryptology*, vol. 1, no. 1, pp. 65–75, 1988.

[84] M. Waidner and B. Pfitzmann, "The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability," in *Advances in Cryptology — EUROCRYPT '89*, 1990, pp. 690–690.

[85] J. Bos and B. den Boer, "Detection of disrupters in the dc protocol," in *Advances in Cryptology — EUROCRYPT '89*, J.-J. Quisquater and J. Vandewalle, Eds., 1990, pp. 320–327.

[86] L. von Ahn, A. Bortz, and N. J. Hopper, "K-anonymous message transmission," in *Proceedings of the 10th ACM SIGSAC CCS*, 2003, pp. 122–130.

[87] P. Golle and A. Juels, "Dining cryptographers revisited," in *Proc. of Eurocrypt 2004*, 2004.

[88] M. Waidner, "Unconditional sender and recipient untraceability in spite of active attacks," in *Advances in Cryptology — EUROCRYPT '89*, 1990, pp. 302–319.

[89] A. Krasnova, M. Neikes, and P. Schwabe, "Footprint scheduling for dining-cryptographer networks," in *FC*, J. Grossklags and B. Preneel, Eds., 2016, pp. 385–402.

[90] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson, "Dissent in numbers: Making strong anonymity scale," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 179–182.

[91] D. Das, S. Meiser, E. Mohammadi, and A. Kate, "Comprehensive anonymity trilemma: User coordination is not enough," *Proceedings on Privacy Enhancing Technologies*, vol. 2020, pp. 356–383, 2020.

[92] H. Corrigan-Gibbs, D. Boneh, and D. Mazieres, "Riposte: An anonymous messaging system handling millions of users," *2015 IEEE Symposium on Security and Privacy*, May 2015. DOI: 10.1109/sp.2015.27. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.27.

[93] L. Barman, M. Zamani, I. Dacosta, J. Feigenbaum, B. Ford, J.-P. Hubaux, and D. Wolinsky, "Prifi: A low-latency and tracking-resistant protocol for local-area anonymous communication," in *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society*, 2016, pp. 181–184.

[94] Z. Newman, S. Servan-Schreiber, and S. Devadas, "Spectrum: High-bandwidth anonymous broadcast," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA: USENIX Association, Apr. 2022, pp. 229–248, ISBN: 978-1-939133-27-4. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/newman.

[95] V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song, "Atlas: Efficient and scalable mpc in the honest majority setting," in *Annual International Cryptology Conference*, Springer, 2021, pp. 244–274.

[96] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell, and A. Nof, "Fast large-scale honest-majority mpc for malicious adversaries," in *Annual International Cryptology Conference*, Springer, 2018, pp. 34–64.

[97] T. Rabin and M. Ben-Or, "Verifiable secret sharing and multiparty protocols with honest majority," in *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, 1989, pp. 73–85.

[98] M. Abspoel, A. Dalskov, D. Escudero, and A. Nof, "An efficient passive-to-active compiler for honest-majority mpc over rings," in *International Conference on Applied Cryptography and Network Security*, Springer, 2021, pp. 122–152.

[99] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1292–1303.

[100] S. Eskandarian, H. Corrigan-Gibbs, M. Zaharia, and D. Boneh, "Express: Lowering the cost of metadata-hiding communication with cryptographic privacy," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1775–1792.

[101] J. Håstad, "The square lattice shuffle," *Random Structures and Algorithms*, vol. 29, no. 4, pp. 466–474, 2006.

[102] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 307–314.

[103] D. L. Shell, "A high-speed sorting procedure," *Communications of the ACM*, vol. 2, no. 7, pp. 30–32, 1959.

[104] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, 1968, pp. 307–314.

[105] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi, "Practically efficient multi-party sorting protocols from comparison sort algorithms," in *Information Security and Cryptology – ICISC 2012*, T. Kwon, M.-K. Lee, and D. Kwon, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 202–216.

[106] A. Aly and S. Cleemput, "An improved protocol for securely solving the shortest path problem and its application to combinatorial auctions," *Cryptology ePrint Archive*, 2017.

[107] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, *et al.*, "Secure multiparty computation goes live," in *International Conference on Financial Cryptography and Data Security*, Springer, 2009, pp. 325–343.

[108] A. Aly, E. Cuvelier, S. Mawet, O. Pereira, and M. V. Vyve, "Securely solving simple combinatorial graph problems," in *International Conference on Financial Cryptography and Data Security*, Springer, 2013, pp. 239–257.

[109] A. Aly, "Network flow problems with secure multiparty computation.," Ph.D. dissertation, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2015.

[110] S. Gao, "A new algorithm for decoding reed-solomon codes," in *Communications, information and network security*, Springer, 2003, pp. 55–68.

[111] B. Kuykendall, H. Krawczyk, and T. Rabin, "Cryptography for #metoo," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 409–429, 2019. DOI: https://doi.org/10.2478/popets-2019-0054. [Online]. Available: https://content.sciendo.com/view/journals/popets/2019/3/article-p409.xml.

[112] P. Mohassel and Y. Zhang, "SecureML: a system for scalable privacy-preserving machine learning," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 19–38.

[113] J. Cartlidge, N. P. Smart, and Y. T. Alaoui, *Mpc joins the dark side*, Cryptology ePrint Archive, Report 2018/1045, https://eprint.iacr.org/2018/1045, 2018.

[114] D. J. Wu, T. Feng, M. Naehrig, and K. Lauter, "Privately evaluating decision trees and random forests," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, 2016.

[115] E. Commission, *2018 reform of eu data protection rules*, May 25, 2018. [Online]. Available: https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf.

[116] E. Goldman, "An introduction to the california consumer privacy act (ccpa)," *Santa Clara Univ. Legal Studies Research Paper*, 2020.

[117] I. Giacomelli, S. Jha, R. Kleiman, D. Page, and K. Yoon, "Privacy-preserving collaborative prediction using random forests," *CoRR*, vol. abs/1811.08695, 2018. [Online]. Available: http://arxiv.org/abs/1811.08695.

[118] M. Keller, *MP-SPDZ: A versatile framework for multi-party computation*, Cryptology ePrint Archive, Report 2020/521, https://eprint.iacr.org/2020/521, 2020.

[119] I. Damgard, V. Pastro, N. Smart, and S. Zakarias, *Multiparty computation from somewhat homomorphic encryption*, Cryptology ePrint Archive, Report 2011/535, https://eprint.iacr.org/2011/535, 2011.

[120] S. Coretti, J. Garay, M. Hirt, and V. Zikas, "Constant-round asynchronous multi-party computation based on one-way functions," in *Advances in Cryptology — ASIACRYPT 2016*, 2016, pp. 998–1021.

[121] J. Bar-Ilan and D. Beaver, "Non-cryptographic fault-tolerant computing in constant number of rounds of interaction," in *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '89, 1989, pp. 201–209.

[122] A. Akavia, M. Leibovich, Y. S. Resheff, R. Ron, M. Shahar, and M. Vald, *Privacy-preserving decision tree training and prediction against malicious server*, Cryptology ePrint Archive, Report 2019/1282, https://eprint.iacr.org/2019/1282, 2019.

[123] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 26–49, 2019.

[124] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1175–1191.

[125] O. Catrina and A. Saxena, "Secure computation with fixed-point numbers," in *Financial Cryptography and Data Security*, R. Sion, Ed., 2010, pp. 35–50.

[126] O. Catrina and S. de Hoogh, "Improved primitives for secure multiparty integer computation," in *Security and Cryptography for Networks*, J. A. Garay and R. De Prisco, Eds., 2010, pp. 182–199.

[127] T. Nishide and K. Ohta, "Multiparty computation for interval, equality, and comparison without bit-decomposition protocol," in *Public Key Cryptography – PKC 2007*, T. Okamoto and X. Wang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 343–360, ISBN: 978-3-540-71677-8.

[128] D. Dua and C. Graff, *UCI machine learning repository*, 2017. [Online]. Available: http://archive.ics.uci.edu/ml.

[129] *Amazon ec2 instance network bandwidth*, https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html.

[130] Á. Kiss, M. Naderpour, J. Liu, N. Asokan, and T. Schneider, "Sok: Modular and efficient private decision tree evaluation," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 2, pp. 187–208, 2019. DOI: https://doi.org/10.2478/popets-2019-0026. [Online]. Available: https://content.sciendo.com/view/journals/popets/2019/2/article-p187.xml.

[131] M. Jones, "Estimating markov transition matrices using proportions data: An application to credit risk," *IMF Working Papers*, vol. 05, Jan. 2006.

[132] H. Muliaman, W. Santoso, B. Santoso, D. Besar, and I. Rulina, "Rating migration matrices: Empirical evidence in indonesia," *IFC Bulletin*, Jan. 2009.

[133] Y. Ishai and E. Kushilevitz, "Randomizing polynomials: A new representation with applications to round-efficient secure computation.," Jan. 2000, pp. 294–304. DOI: 10.1109/SFCS.2000.892118.

[134] P. Mohassel and M. Franklin, "Efficient polynomial operations in the shared-coefficients setting," 2006, pp. 44–57. DOI: 10.1007/11745853_4.

[135] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung, "Secure efficient multiparty computing of multivariate polynomials and applications," 2011, pp. 130–146. DOI: 10.1007/978-3-642-21554-4_8.

[136] R. Cramer and I. Damgård, "Secure distributed linear algebra in a constant number of rounds," in *CRYPTO*, 2001, pp. 119–136. DOI: 10.1007/3-540-44647-8_7.

[137] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel, "Privacy-preserving remote diagnostics," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 498–507.

[138] M. Joye and F. Salehi, "Private yet efficient decision tree evaluation," in *IFIP Annual Conference on Data and Applications Security and Privacy*, Springer, 2018, pp. 243–259.

[139] A. Tueno, F. Kerschbaum, and S. Katzenbeisser, "Private evaluation of decision trees using sublinear cost," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 1, pp. 266–286, 2019.

[140] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev, "New primitives for actively-secure mpc over rings with applications to private machine learning," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1102–1120. DOI: 10.1109/SP.2019.00078.

[141] Y. Ishai, E. Kushilevitz, S. Meldgaard, C. Orlandi, and A. Paskin-Cherniavsky, "On the power of correlated randomness in secure computation," 2013, pp. 600–620. DOI: 10.1007/978-3-642-36594-2_34.

[142] S. Wagh, D. Gupta, and N. Chandran, "SecureNN: 3-party secure computation for neural network training," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 26–49, Jul. 2019. DOI: 10.2478/popets-2019-0035.

[143] J.-L. Watson, S. Wagh, and R. A. Popa, *Piranha: A gpu platform for secure computation*, Cryptology ePrint Archive, Paper 2022/892, https://eprint.iacr.org/2022/892, 2022. [Online]. Available: https://eprint.iacr.org/2022/892.