RPM: Robust Anonymity at Scale

Donghang Lu
Purdue University
West Lafayette, IN, USA
lu5621994@gmail.com

Aniket Kate
Purdue University / Supra
West Lafayette, IN, USA
aniket@purdue.edu

ABSTRACT

This work presents RPM, a scalable anonymous communication protocol suite using secure multiparty computation (MPC) with the offline-online model. We generate random, unknown permutation matrices in a secret-shared fashion and achieve improved (online) performance and the lightest communication and computation overhead for the clients compared to the existing robust anonymous communication protocols. Using square-lattice shuffling, we make our protocol scale well as the number of clients increases. We provide three protocol variants, each targeting different input volumes and MPC frameworks/libraries. Besides, due to the modular design, our protocols can be easily generalized to support more MPC functionalities and security properties as they get developed. We also illustrate how to generalize our protocols to support two-way anonymous communication and secure sorting. We have implemented our protocols using the MP-SPDZ library suit. The benchmark demonstrates that our protocols achieve unprecedented online phase performance with practical offline phases.

KEYWORDS

Secure Multiparty Computation, Anonymous Communication, Secure Random Permutation

1 INTRODUCTION

There are by now millions of users using the Tor network [27, 28] to break the link between their identities and their messages/packets. As the solutions like Tor network suffer from traffic analysis attacks [11, 31, 38], anonymous communication becomes an active research area and many works [1, 3, 29, 42, 43, 47, 52, 53] aim at providing anonymous communication services efficiently. This work explores 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 [1, 47]. 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 [20, 22]. Meanwhile, clients with limited computation power can benefit from our protocol significantly.

RPM employs the standard offline/online model of MPC [7, 23, 37, 46, 47], 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 broadcast can be achieved by performing a random permutation to the input messages. Recall that for any k-input permutation π, there exists a zero-one matrix M_π such that $\pi(x) = M_\pi x$. Therefore, if we obtain a permutation matrix such that anyone including the adversary does not know the underlying permutation, the 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 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). Moreover, 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 [39], 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

[1] RPM stands for Random permutation matrix.

Proceedings on Privacy Enhancing Technologies 2023(2), 347–360

https://doi.org/10.56553/popets-2023-0057

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 per server. 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× more time in the worst cases.

1.1 Paper Organization

We introduce the related works in Section 2. We present the system model of our protocols and introduce the background and preliminaries in Section 3. We explain the detailed construction of our protocols in Section 4. We compare the theoretical complexities of our protocols with related works in Section 4.7. In Section 5.1, we show how our protocols are implemented using MP-SPDZ framework and the detailed benchmark data is available in Section 5.2. Besides, we illustrate the construction and the performance of the robust version of our protocols in Section 5.4. In Section 6, we introduce how to apply our protocols to solve more higher-level applications.

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 [25] as well as empirically [11, 31, 38].

Mixing networks (mixnets) [18] 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 [17, 24, 42–45, 52, 53] have been proposed that can deter traffic analysis attacks; however, their high latency overheads of several seconds (at least) are 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) [16] and its successors [14, 33, 41, 54–57] are much better suited. Using a cryptographic setup/coordination among clients, these schemes offer provably strong anonymity in a constant number of rounds [26]. 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 on the client-side, the idea of employing some MPC servers is getting popular [1, 3, 8, 21, 47]: 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 [47], Blinder [1], and Clarion [29] to be the closest to our work.

AsynchroMix [47] 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 (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 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 [1] 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 communication cost and to deal with 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 [21] uses similar approaches of offloading part of the computation to the client-side. It uses discrete point functions to help reduce client communication costs and achieves the same client-side complexity as Blinder.

Eskandarian and Boneh propose a protocol called Clarion [29], which is communication-efficient to do anonymous broadcast. They propose constructions for both 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 [47], we will get exactly the same security

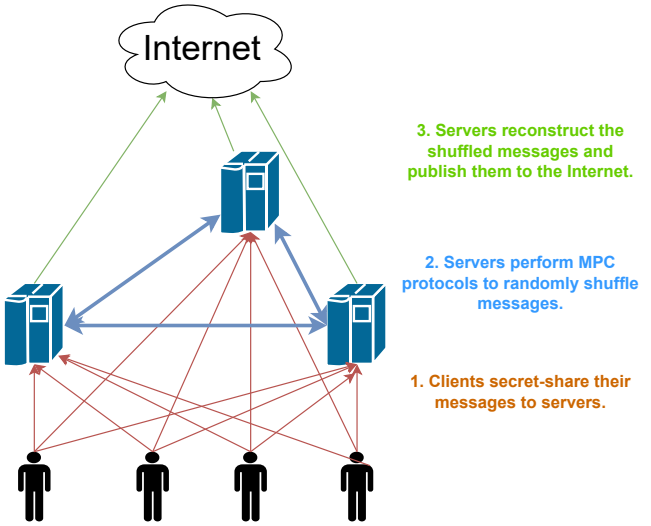

Figure 1: Client-server setting for anonymous communication with MPC shuffling.

properties as Asynchromix. However, it is impossible to do so for Clarion.

There are also works focusing on specialized applications. Spectrum [49] 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 PRELIMINARY

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 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 [1, 47], 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.

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 based 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.

3.2 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.

Non-goals.We list the non-goals below:

- **Confidentiality**: Our protocols do not aim at protecting 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 Secret-sharing-based MPC

3.3.1 Shamir Secret Sharing. Shamir secret sharing scheme with threshold (n, t), where $n > t \geq 0$, allows the dealer to share a secret $s \in F_p$ to n parties $\{P_1, \ldots, P_n\}$ such that the s is revealed if and only if $t+1$ or more parties combine their shares to reconstruct the secret value. To share a secret s, the dealer samples a degree-t polynomial $\phi()$ such that the constant coefficient of $\phi()$ is the secret s, and all other coefficients are set to be random elements. Then the dealer sends the share $\phi(i)$ to the party P_i. We denote $[\![s]\!]_t^i$ as the secret share of party P_i for the rest of the paper. We may omit the superscript/subscript of a share when it is clear from the context.

To reconstruct the secret, parties send their private shares to each other. When the party gathers the shares from $t+1$ parties, it is sufficient for it to reconstruct the polynomial $\phi()$ and the secret s can be computed through $s = \phi(0)$.

3.3.2 MPC with Shamir Secret Sharing. In this work, we focus on a client-server setting where clients secret-share their private inputs to a group of servers, and the servers perform MPC protocols collaboratively. For the arithmetic-circuit MPC, a computed functionality can be represented using the addition gates and multiplication gates. Shamir secret sharing is additive homomorphic in the sense that the following equation holds:

$$[\![a+b]\!]_t = [\![a]\!]_t + [\![b]\!]_t$$

Therefore, any addition and the linear combination of the secret values can be performed locally by applying the same operations over the shares. However, when it comes to multiplication, the multiplication of two degree-t polynomials results in degree-$2t$ polynomials. Thus it cannot be achieved locally and we often follow the online/offline MPC paradigm here and use Beaver triples [12] to do the multiplication. As a result, the multiplication becomes an interactive protocol among servers and requires communication. When measuring the complexity of an MPC protocol, we usually think of additions as free and only consider the number/rounds of multiplications as they are the bottleneck of the protocols in most cases.

Offline/Online Model. We often separate an MPC protocol into an input-independent offline phase and an input-dependent online phase. In the offline phase, servers prepare the input-independent secret shares such that they can be consumed in the online phase to make the online phase faster. The offline phase can be run for a long time before the actual online phase starts, therefore it is allowed that the offline phase is more costly than the online phase. The starting point of this work is to properly design the offline phase protocols and the online phase protocols such that most cryptographical expensive operations are moved to the offline phase, and the online phase only requires some basic operations such as reconstructions. As the offline phase is often more costly than the online phase, a standard workflow is as follows: Before the online protocol starts, the servers can run the offline phase in advance, which may take a long time to finish. After that, the clients send their input to the server to execute the efficient online phase.

The goal of RPM is to optimize the online phase performance. As the offline phase is input-independent, the offline phase can be run early in advance, even days before the actual online phase. This is especially useful when computations are done regularly but not continuously. For example, an anonymous broadcasting system can run the offline phase during the night and run the online phase during the day once it receives user input. Therefore, it is the online phase that actually determines the user experience. The users will not be influenced by a more costly but still practical offline phase.

3.4 Beaver Triple Multiplication [12] for Scalars and Matrices

To multiply two secret shares $[\![x]\!]$ and $[\![y]\!]$, the servers prepare a precomputed triple $[\![a]\!]$, $[\![b]\!]$, $[\![ab]\!]$ where a and b are random elements. In the online phase, servers compute and reconstruct $(x-a)$ and $(y-b)$, then the result is shown as follows:

$$[\![xy]\!] = (x-a)(y-b) + (x-a)[\![b]\!] + (y-b)[\![a]\!] + [\![ab]\!]$$

The equation above only involves the linear combination of secret sharing, and thus can be computed locally. Therefore, the cost of Beaver Multiplication is two reconstructions in one round.

Beaver's technique naturally extends to the multiplication of two secret-shared matrices. The only change to the steps is to replace the single elements with matrices [48]. The communication complexity of multiplying two k-by-k secret shared matrices is $O(k^2)$, because it requires the reconstructions of two k-by-k matrices. It is more efficient than simply using $O(k^3)$ beaver multiplications to compute each cell. In the rest of this paper, we use the extended Beavers idea when we refer to the multiplication of two secret-shared matrices.

3.5 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 [47] 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 of 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.

3.6 Notations

We summarize notations that appear in the rest of the paper 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.

4 USING PERMUTATION MATRICES FOR ANONYMOUS COMMUNICATION

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 [19, 34, 58] (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.

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 valid (n, t) secret sharing. To solve this problem, we use a similar method used in [47]: 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 computes 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 $O(n)$.

4.2.1 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.

4.3 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 [2, 40, 47, 50] 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 [2, 7] 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.

4.4 The First Variant

4.4.1 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 to 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 [46].

In the malicious setting, we also need to guarantee that the matrices shared by servers are indeed 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 [1, 15, 30]. To verify a vector $w = (w_1, \ldots, w_k)$, the sketch is represented by $(\sum_{i=1}^{k} w_i \cdot r_i)^2 - m(\sum_{i=1}^{k} w_i r_i^2)$ 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 each of the permutation matrices and both properties can be properly verified. The cost of this check is cheap since each sketch only includes one secret sharing multiplication. The offline phase is summarized in Algorithm 2.

Algorithm 2: The offline phase of Variant 1

```
for i ← 1 to t + 1 do
    Server P_i generates a k-by-k permutation matrix M_i and
      secret-share it to all servers
    for i ← 1 to k do
        All servers perform sketch checks mentioned in
          Section 4.4.1 on the i-th row and i-th column of
          [[M_i]].
        If any check fails, abort.
All Servers multiply and compute
  [[P]] = [[M_1]][[M_2]] ... [[M_{t+1}]]
Output [[P]].
```

4.4.2 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 [7, 19, 58] 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 $X = \{x_1, x_2, \ldots, x_k\}$ and $Y = \{y_1, y_2, \ldots, y_k\}$:

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 $i \leftarrow 1$ *to* k **do**
  Y[i] = Inner-product-and-reconstruct($[\![P[i]]\!]$, $[\![X]\!]$)
Output Y.

4.5 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.

4.5.1 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
  for $i \leftarrow 1$ *to* k **do**
    All servers perform sketch checks mentioned in Section 4.4.1 on the i-th row and i-th column of $[\![M_i]\!]$.
    If any check fails, abort.
All Servers multiply and compute $[\![P]\!] = [\![M_1]\!][\![M_2]\!] \ldots [\![M_{t+1}]\!]$
All servers generate k random shares $[\![R]\!] = \{[\![r_1]\!], [\![r_2]\!], \ldots, [\![r_k]\!]\}$.
All servers compute $[\![PR]\!] = Mul([\![P]\!], [\![R]\!])$.
Output $[\![P]\!]$, $[\![PR]\!]$, $[\![R]\!]$.

Online Phase. Given the input messages vector $X = \{x_1, \ldots, x_k\}$ 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 : $[\![X]\!] = \{[\![x_1]\!], \ldots, [\![x_k]\!]\}$
Output : Y
Pre-computation : $[\![P]\!]$, $[\![R]\!]$, $[\![PR]\!]$

$[\![X + R]\!] = [\![X]\!] + [\![R]\!]$
$X + R = Open([\![X + R]\!])$
$[\![Y]\!] = [\![P]\!] \cdot (X + R) - [\![PR]\!]$
$Y = Open([\![Y]\!])$

4.6 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[3, 36]. In [36], 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 Figure 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)

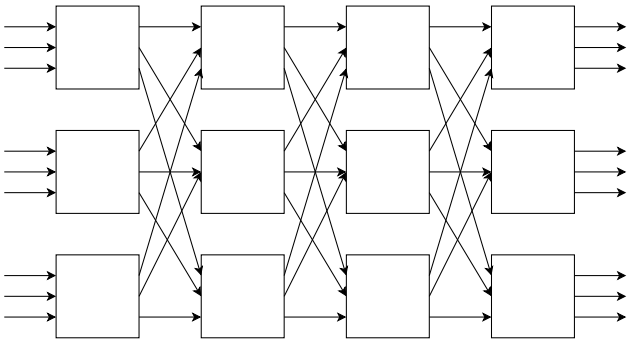

Figure 2: An illustrative example of a square network with $k = 9$. The network consists of $q = 4$ layers, where each layer has $\sqrt{k} = 3$ permutation nodes, represented by square blocks. Each permutation node takes $\sqrt{k} = 3$ messages as inputs, randomly permute and output them. In Variant 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.

4.7 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 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 [47], which has the closest online phase communication and rounds. The communication cost of the 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.

4.8 Security Analysis

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

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

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.

Table 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 [3]	$O(1)$	$O(1)$	$O(\alpha k \log k)^*$	$O(\alpha k \log k)^*$	$O(\log k)$	$n = 3, t = 1$	✗
Switching Network [47]	$O(1)$	$O(n)$	$O(k \log^2 k)$	$O(k \log^2 k)$	$\log^2 k$	$n \geq 3t + 1$	✓
PowerMix [47]	$O(1)$	$O(n)$	$O(k)$	$O(k^3)$	2	$n \geq 3t + 1$	✓
Blinder [1]	$O(\sqrt{k})$	$O(n \cdot \sqrt{k})$	$O(k)$	$O(k^2)$	$O(1)$	$n \geq 4t + 1$	✓
Clarion [29]	$O(1)$	$O(n)$	$O(k)$	$O(k)$	$O(n)$	$n > t$	✗
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 restrictions might be needed to support more security properties (e.g., $n \geq 3t + 1$ for robustness).

* In McMix [3], 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.

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 first provide the general idea here and put the complete proof in Section 4.9. 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 the 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 from 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.

4.9 Security Proof

We first provide the ideal functionality that our protocols achieve, then we present a simulator-based proof to prove the security. The multiparty random permutation ideal functionality is inspired by Clarion [29] and we modify it to fit our protocols better. We model the robustness as an optional security property, and the ideal functionality does not include it.

Definition 4.1. (Ideal Functionality of Secure Random Permutation) A secure random permutation functionality **F** interact with n servers $P_1, P_2, \ldots, P_n$ and k' clients $C_1, C_2, \ldots, C_{k'}$ ($k \leq k'$, where k is the number of messages mixed in each run). We assume the existence of an adversary that can control at most $k - 2$ clients and $t < \frac{n}{2}$ servers. Any server controlled by the adversary could send an **abort** to **F** at any time, which leads to a protocol abort.

F initiates an empty array **T** and waits for messages from clients. Any message received will be added to **T**. After collecting k inputs $M = \{m_1, m_2, \ldots, m_k\}$, **F** secret share the vector M with (n, t) secret sharing, and send the shares to all the servers, therefore the adversary will have t shares for each input message. Next, **F** runs the permutation step as follows: **F** samples a random permutation π and compute $M' = \pi(M)$. Then **F** sends M' together with the secret sharing of M' to the adversary, the adversary could respond with either **finish** or **abort**. If the response is **finish**, **F** sends M' to all the servers and outputs M'. If the response is **abort**, the protocol aborts.

Theorem 1. Assuming the existence of a malicious secure MPC framework, especially, let **Open** be the malicious secure secret sharing reconstruction, let **DotProduct** be the malicious secure inner product, RPM realizes the ideal functionality **F** defined in Definition 4.1.

Proof Sketch. We build up a simulator S to simulate the view of the adversary. Without loss of generality, we assume the worst case where the adversary controls all but two honest clients, and t servers $P_1, P_2, \cdots, P_t$.

The first step of the ideal functionality is to collect input messages from clients in the form of secret sharing. There are k input messages in total. For each message, the adversary has access to t shares as it controls t malicious servers. To simulate the view of the adversary, for each input message in M, the simulator **S** can generate t random values as t shares. This is identical to the adversary's view because any t shares leak no information about the secret message.

Next, **F** perform the random permutation to the input message vector M. Here the view of the adversary includes the intermediate communications when **F** executes the shuffling, and we show how to simulate them with **S** below:

In Variant 1, the mixing steps only include doing k secure inner products. Therefore, the malicious security of our protocol is reduced to the security of **DotProduct**. In Variant 2, the mixing steps require the reconstruction of $[\![M + R]\!]$. As R is random, $M + R$ is also uniformly random. Therefore **S** can directly generate k random values and send them to all the servers as the simulation of $M + R$. The malicious security of the reconstruction step depends on the malicious secure building block **Open**. For Variant 3, the mixing step is a combination of multiple instances of Variant 1 or Variant 2. The output of a permutation node is directly taken as the input of the permutation nodes in the next layer, no more information is leaked outside of permutation nodes. Therefore **S** can simulate the view of the adversary using the strategy the methods above.

Finally, the parties reconstruct the output messages M' through secret sharing reconstruction. The view of the adversary includes the final output messages M', and the corresponding n secret shares of each message. For each opened message m, The simulator **S** has access to m and t shares as those t shares are stored in t malicious servers. The goal of **S** is to generate the rest $n - t$ shares for each message m, which can be done depending on the underlying secret sharing schemes. We take Shamir secret sharing as an example. For each message, **S** has t points and the secret s, which can also be treated as a point $(0, s)$. **S** can use these $t + 1$ points to uniquely decide a degree-t polynomial, and use this polynomial to compute the rest of the shares.

After the reconstruction, the adversary can decide to continue or abort. If the adversary decides to continue, the protocol outputs M' and finishes. □

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

There are currently many MPC libraries [34, 39, 47, 58] available with different trade-offs. Among them, we choose to use MP-SPDZ [39] 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 available in a public GitHub repo[2].

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.

5.2 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).

Variant 1. First, we present the benchmark result of Variant 1. The result is available in Table 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.

Variant 2. The benchmark numbers of Variant 2 are available in Table 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 a fast inner product.

[2]the link: https://github.com/lu562/MP-SPDZ

Table 2: Performance of the online phase of Variant 1 **in the 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 the 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

As a result, we also use Variant 2 as the building blocks to conduct the benchmark of Variant 3.

Table 3: Performance of the online phase of Variant 2 **in the three-party setting. (k refers to the number of clients. message size is** 16 **bytes. Communication is measured by the 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 4. The result shows that we can permute $k = 90000$ messages in around 12 seconds with 46MB communication.

Table 4: Performance of the online phase of Variant 3 **in the 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 the 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 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 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

5.3 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 6: Performance of the offline phase of Variant 1 **in the three-party setting. (k refers to the number of clients. Communication is measured by the total MB sent per party)**

k	Offline Time (s)	Online Communication(MB)
1000	5.1	32
3000	126	288
5000	615	800
7000	1832	1568
10000	5767	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 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 7: Offline phase benchmark for Variant 3**. (k is the number of clients, communication is measured by the total MB sent per server)**

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

5.4 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 in Section 4, 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 [47] and construct a robust Shamir secret-sharing reconstruction using Reed-Solomon decoding [32]. 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 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 [32]. 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 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 8. In worst cases, the performance is approximately 2× more than the non-robust version.

Table 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 the wrong shares to simulate the worst case.**

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

5.5 Performance Comparison

We present a comparison to PowerMix [47], Blinder [1], and Clarion [29] as they share the closest theoretical complexity with ours.

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 that online computation is the main bottleneck of the whole protocol.

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 Blinder's client cost is $O(\sqrt{k})$.

For Clarion [29], 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.

6 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.

6.1 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 [1, 3, 29], where senders and receivers conduct private conversations such that the adversary has no information about their identities.

Inspired by [18], our 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,

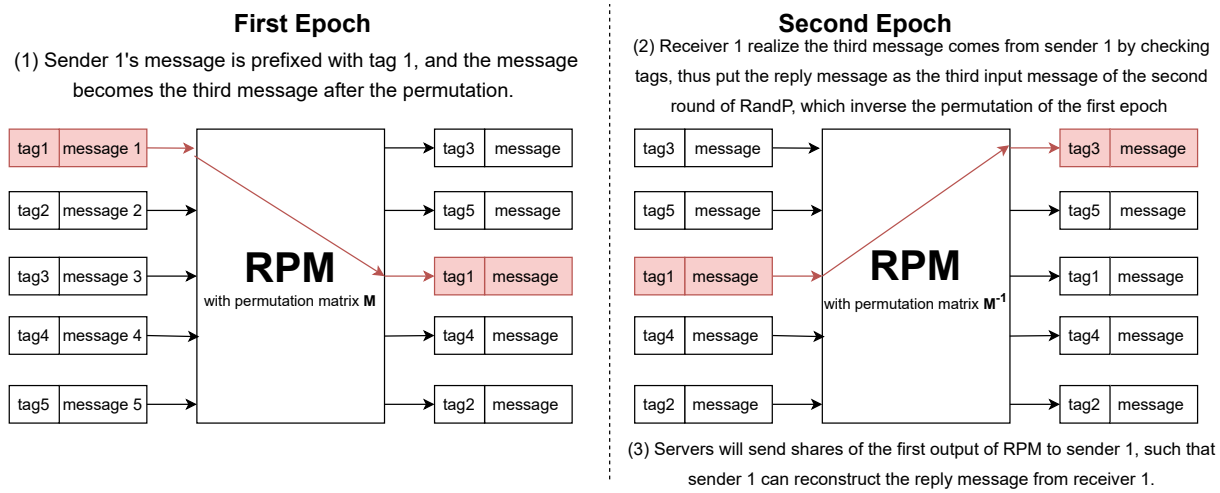

Figure 3: An example of two-way communication. The example includes five participating clients, and we mark the protocol flow for the first sender (denoted as sender 1) in red. Sender 1 and the corresponding receiver agree with a tag (tag1 in the figure) before the protocol.

senders and receivers agree on some tags offline[3], such that these tags can be prefixed to the sender's 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.

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 [1, 3].

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.

[3]This step is out of the scope of this paper. Establishing shared secrets is a well-studied problem and a lot of prior works could be used to achieve it.

6.2 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 [9, 10, 51]. However, to the best of our knowledge, the most practical oblivious sorting is achieved by sorting networks [10] with $O(k \log^2 k)$ communication in $O(\log^2 k)$ rounds.

Recently, Hamada et, al [35] propose to use data-dependent algorithms in an oblivious fashion to solve the sorting problem. The idea is that parties can perform a secure random shuffle to the input, reconstruct the inputs, then compute data-dependent algorithms locally with the reconstructed input. With this idea, the secure sorting problem is reduced to a 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 [5, 13], combinatorial graph problems [6], and network flow problems [4]. Therefore, our protocols can be beneficial to much more applications than just anonymous communication.

7 CONCLUSION

In this work, we build up protocols for efficient random permutation, and use them to achieve anonymous communication by randomly permuting the messages. We perform three variants of our protocols, each targeting different MPC frameworks and real-world applications. The benchmark illustrates that our protocols are efficient in both the online phase and the offline phase, besides, the client cost of our protocol is the lowest among all existing works, which makes our protocols friendly to clients with limited networks and computation power. Finally, as we employ MPC in a generic fashion, our performance will improve further as better MPC protocols and libraries get developed in the near future.

For future works, we find the implementation of our work has great potential to be improved. First, the use of GPU can significantly improve the performance of our protocols as the local computation in our protocols is highly parallelizable. Besides, we want to explore the possibility to make the offline phase more efficient. Considering the permutation matrix consists of only zeros and ones, our offline phase could potentially be more efficient using binary circuits, and it could finally produce a mixed-circuit solution to make both the offline phase and the online phase efficient.

ACKNOWLEDGMENTS

This work has been partially supported by the National Science Foundation (NSF) under grant CNS-1846316. We thank Marcel Keller for the detailed guidance about MP-SPDZ backends and low-level constructions. We thank Andrew Miller for the valuable discussion about robust secret-sharing reconstruction.

REFERENCES

[1] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder–scalable, robust anonymous committed broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1233–1252, 2020.

[2] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority mpc over rings. In *International Conference on Applied Cryptography and Network Security*, pages 122–152. Springer, 2021.

[3] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, Vancouver, BC, August 2017. USENIX Association.

[4] Abdelrahaman Aly. *Network flow problems with secure multiparty computation*. PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2015.

[5] Abdelrahaman Aly and Sara Cleemput. An improved protocol for securely solving the shortest path problem and its application to combinatorial auctions. *Cryptology ePrint Archive*, 2017.

[6] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In *International Conference on Financial Cryptography and Data Security*, pages 239–257. Springer, 2013.

[7] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 695–712, 2018.

[8] Ludovic Barman, Mahdi Zamani, Italo Dacosta, Joan Feigenbaum, Bryan Ford, Jean-Pierre Hubaux, and David 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*, pages 181–184, 2016.

[9] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[10] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[11] K. S. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. C. Sicker. Low-resource routing attacks against tor. In *WPES'07*, pages 11–20, 2007.

[12] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO 1991*, pages 420–432, 1992.

[13] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[14] Jurjen Bos and Bert den Boer. Detection of disrupters in the dc protocol. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology — EUROCRYPT '89*, pages 320–327, 1990.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[16] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.

[17] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri de Ruiter, and Alan T. Sherman. cmix: Mixing with minimal real-time asymmetric cryptographic operations. In *ACNS*, 2017.

[18] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[19] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

[20] Michele Ciampi, Muhammad Ishaq, Malik Magdon-Ismail, Rafail Ostrovsky, and Vassilis Zikas. Fairmm: A fast and frontrunning-resistant crypto market-maker. *Cryptology ePrint Archive*, 2021.

[21] Henry Corrigan-Gibbs, Dan Boneh, and David Mazieres. Riposte: An anonymous messaging system handling millions of users. *2015 IEEE Symposium on Security and Privacy*, May 2015.

[22] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234*, 2019.

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

[24] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: design of a type iii anonymous remailer protocol. In *2003 Symposium on Security and Privacy, 2003.*, pages 2–15, 2003.

[25] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Anonymity trilemma: Strong anonymity, low bandwidth overhead, low latency-choose two. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 108–126. IEEE, 2018.

[26] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Comprehensive anonymity trilemma: User coordination is not enough. *Proceedings on Privacy Enhancing Technologies*, 2020:356–383, 2020.

[27] 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.

[28] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, page 21, 2004.

[29] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. *Cryptology ePrint Archive*, 2021.

[30] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1775–1792, 2021.

[31] N. S. Evans, R. Dingledine, and C. Grothoff. A Practical Congestion Attack on Tor Using Long Paths. In *USENIX'09*, pages 33–50, 2009.

[32] Shuhong Gao. A new algorithm for decoding reed-solomon codes. In *Communications, information and network security*, pages 55–68. Springer, 2003.

[33] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *Proc. of Eurocrypt 2004*, 2004.

[34] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. Atlas: efficient and scalable mpc in the honest majority setting. In *Annual International Cryptology Conference*, pages 244–274. Springer, 2021.

[35] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *Information Security and Cryptology – ICISC 2012*, pages 202–216, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[36] Johan Håstad. The square lattice shuffle. *Random Structures and Algorithms*, 29(4):466–474, 2006.

[37] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. On the power of correlated randomness in secure computation. In

Theory of Cryptography Conference, pages 600–620. Springer, 2013.
[38] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. Users get routed: Traffic correlation on tor by realistic adversaries. In *Proc. ACM SIGSAC conference on Computer & communications security 2013*, pages 337–348, 2013.
[39] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.
[40] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, 2016. Association for Computing Machinery.
[41] Anna Krasnova, Moritz Neikes, and Peter Schwabe. Footprint scheduling for dining-cryptographer networks. In Jens Grossklags and Bart Preneel, editors, *FC*, pages 385–402, 2016.
[42] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 406–422, 2017.
[43] David Lazar, Yossi Gilad, and Nickolai 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*, pages 711–725, 2018.
[44] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A Scalable, Traffic Analysis Resistant Anonymity Network for VoIP Systems. In *Proc. ACM SIGCOMM 2015*, pages 639–652, 2015.
[45] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards Efficient Traffic-analysis Resistant Anonymity Networks. In *Proc. ACM SIGCOMM 2013*, pages 303–314, 2013.
[46] Donghang Lu, Albert Yu, Aniket Kate, and Hemanta Maji. Polymath: Low-latency mpc via secure polynomial evaluations and its applications. *Proceedings on Privacy Enhancing Technologies*, 2022(1):396–416, 2022.
[47] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew 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*, pages 887–903, 2019.
[48] P. Mohassel and Y. Zhang. SecureML: a system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 19–38, 2017.
[49] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 229–248, Renton, WA, April 2022. USENIX Association.
[50] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, 1989.
[51] Donald L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2(7):30–32, 1959.
[52] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.
[53] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.
[54] Luis von Ahn, Andrew Bortz, and Nicholas J. Hopper. K-anonymous message transmission. In *Proceedings of the 10th ACM SIGSAC CCS*, page 122–130, 2003.
[55] Michael Waidner. Unconditional sender and recipient untraceability in spite of active attacks. In *Advances in Cryptology — EUROCRYPT '89*, pages 302–319, 1990.
[56] Michael Waidner and Birgit Pfitzmann. The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability. In *Advances in Cryptology — EUROCRYPT '89*, pages 690–690, 1990.
[57] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.
[58] Yihua Zhang, Aaron Steele, and Marina Blanton. Picco: a general-purpose compiler for private distributed computation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 813–826, 2013.

A BENCHMARK OF TWO-WAY COMMUNICATION

The benchmark result is shown in Table 9. As the protocol is essentially two executions of the Variant 3, we see that the performance numbers are also approximately 2× of the numbers in Variant 3 benchmark.

Table 9: Performance of the two-way communication. (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	1.08	9.92
40000	7.89	39.68
90000	25.95	89.28
160000	57.22	158.72