Rabbit-Mix: Robust Algebraic Anonymous Broadcast from Additive Bases

Chongwon Cho[1], Samuel Dittmer[1], Yuval Ishai[2], Steve Lu[1], and Rafail Ostrovsky[3]

[1]Stealth Software Technologies Inc., USA
[2]The Computer Science Department, Technion, Israel
[3]The Computer Science and Mathematics Department, UCLA, USA

Abstract

We present Rabbit-Mix, a robust algebraic mixing-based anonymous broadcast protocol in the client-server model. Rabbit-Mix is the first practical sender-anonymous broadcast protocol satisfying both robustness and 100% message delivery assuming a (strong) honest majority of servers. It presents roughly $3\times$ improvement in comparison to Blinder (CCS 2020), a previous anonymous broadcast protocol in the same model, in terms of the number of algebraic operations and communication, while at the same time eliminating the non-negligible failure probability of Blinder. To obtain these improvements, we combine the use of Newton's identities for mixing with a novel way of exploiting an algebraic structure in the powers of field elements, based on an *additive 2-basis*, to compactly encode and decode client messages. We also introduce a simple and efficient distributed protocol to verify the well-formedness of client input encodings, which should consist of shares of multiple arithmetic progressions tied together.

1 Introduction

Anonymous communication is one of the most central computer security goals, yet it is also among the most difficult to realize at scale. Anonymous communication should hide the identities of senders and recipients even against network surveillance and traffic analysis. Some of the best-known practical approaches for the problem include Tor [16] and anonymization proxies [19, 26, 30]. However, these approaches are vulnerable to adversaries with powerful traffic analysis capabilities, which potentially control a portion of anonymous network servers [24, 25, 37].

Leveraging MPC A cryptographic approach for overcoming traffic analysis leverages secure multiparty computation (MPC) [6, 11, 21] to build anonymous communication protocols. Chaum [10] introduced the *Dining Cryptographers* (DC) problem formalizing the cryptographic notion for anonymity, and proposed a seminal MPC-based solution for sender-anonymous broadcast (AB). In his approach, referred to as a DC-network (DC-net in short), a group of players can publish their messages without revealing the information of linkage between the players and their messages. The main advantage demonstrated by the DC-net approach was the strong anonymity guarantee even against adversaries using powerful adversarial traffic analysis as long as the players behave honestly and communicate over authenticated point-to-point (P2P) channels. A main drawback of the original DC-net approach is its poor scalability. Every player must have secure communication channels with all other players, and the number of players can be potentially large. A second drawback is that the system is easily disrupted by malicious players.

AB in the Client-Server Model To address both of the above limitations, a large number of of works considered scalable DC-net protocols in the so-called "client-server" model. In this model, a small set of N_s servers jointly provides a sender-anonymous virtual bulletin board to N_c clients[1], where $N_s \ll N_c$ [1, 2, 12–14, 17, 18, 27, 38, 40]. In particular, Dissent [13, 40], Verdict [14], and Riposte [12] proposed AB protocols with $O(N_c N_s)$ overall communication showing off better scalability than typical client-to-client DC-net protocols that have $O({N_c}^2)$ overall communication. Dissent built a DC-net in the client-server model and Verdict is built upon Dissent, and uses zero-knowledge proofs of knowledge to prevent disruption from malicious clients. Riposte and Express [18] rely on Function Secret Sharing (FSS) [8], which in turn only requires symmetric cryptography (e.g. AES). Sabre [38] inherits the structure of Riposte and Express and improves by employing fast secret-shared non-interactive proofs/arguments(of Knowledge) for more efficient audits on client messages. Indeed, Riposte presented a seminal blueprint for AB in which message shuffling is achieved via *private-writing* (PW). MCMix [2] presented a 3-server shuffling protocol

[1]Here, N_c is an upper-bound on the size of the anonymity set rather than the actual number of clients. See Section 1 for further discussion

with a different shuffling paradigm. Clarion [17] proposed a N_s-server shuffling protocol which improves over MCMix. We note that all protocols mentioned above do not guarantee resilience against malicious servers. Concretely, even a single misbehaving server can mount a Denial-of-Service attack by preventing output delivery, and can potentially weaken the security guarantee by disqualifying a subset of the clients, effectively reducing the size of the anonymity set.

Resisting Malicious Servers Lu et al. [27] pointed out that *fairness* of the underlying MPC protocol is essential to mitigate the leakage of information linking clients and their messages to corrupted servers. Fair MPC protocols guarantee that an adversary obtains protocol outputs only if honest clients receives their outputs. If an AB protocol is unfair, then an adversary may receive output messages and terminate the protocol forcing potentially some clients to participate again in the next epoch with the same messages.

Motivated by this security concern, Lu et al. [27] constructed the first practical AB protocols with *fair output delivery* in an asynchronous network by relying on honest-majority MPC based on Shamir's secret sharing. More concretely, they proposed two distinct mixing protocols: AsynchroMix and PowerMix. AsynchroMix uses honest-majority MPC to realize the butterfly sorting network for message shuffling. It has $O(1)$ client-to-server (c-to-s) communication and $O(N_s N_c \log^2 N_c)$ server-to-server (s-to-s) communication and computation with $O(\log^2 N_c)$ rounds. PowerMix is based on Newton's identities relating the sums of powers and symmetric polynomials, where the roots of the symmetric polynomial are the client messages [12, 27, 28, 33]. PowerMix is light on the client side: it has $O(1)$ c-to-s communication and client computation. However, it has $O({N_c}^3)$ server computation and $O({N_c}^2)$ (s-to-s) communication. It also requires an expensive offline protocol for securely generating a large amount of secret correlated randomness. AsynchroMix and PowerMix both do not guarantee resiliency against DoS attacks by corrupted servers since corrupted servers may choose to disrupt the mixing protocols at the cost of receiving no client message.

To address this limitation, Abraham, Pinkas, and Yanai [1] presented Blinder, the first practical AB protocol with full *robustness* (i.e., guaranteed output delivery). Following the blueprint of Riposte [12], Blinder also uses a PW-based approach to shuffle messages by writing them into random locations, but realizes this approach using an honest-majority MPC protocol based on Shamir's secret sharing. Due to the nature of the PW approach, Blinder requires communication and computation redundancy to (1) minimize the probability of severe collisions (e.g., messages will be lost if more than two clients write into the same location) and (2) recover messages from mild collisions if a collision involves two client messages. Similarly to Riposte, Blinder achieves the first goal by increasing the size of the PW database by some constant factor c and the second goal by using two database tables and a simple algebraic technique for recovering (at most) two colliding messages. As a result, it inherits Riposte's 95% message delivery success rate for $c = 2.75$, in contrast to PowerMix's 100% delivery success rate. Blinder has roughly $2c\sqrt{N_c}$ client-to-server communication and $O({N_c}^2)$ server computation/communication.

Technical Goal and Approach The goal of this work is to build a practical AB protocol that simultaneously achieves *robustness* (i.e., guaranteed output delivery) and *100%* message delivery success rate, while concretely improving both communication and computation complexities over the state-of-art MPC-based AB protocols.

This goal is achieved by a new robust AB protocol that we call Rabbit-Mix that we build upon authenticated point-to-point channels and broadcast channels. Our main technical approach is to combine Newton's identity based shuffling technique with novel techniques for re-balancing and optimizing communications and computations by exploiting the algebraic structure of powers of messages. As discussed above, AB protocols that follow a PW-based approach inherently require redundant blowups in their communications and computations in order to reduce their message delivery failure rates. We mitigate this redundancy while achieving perfect message delivery by using Newton's identities as a main building block, similarly to PowerMix. However, PowerMix's server computation requires $O({N_c}^3)$ algebraic operations to privately compute secret shares of N_c consecutive powers (e.g., $m, m^2, \cdots, m^{N_c}$) of each client's message m, where PowerMix clients submit a single secret-share of their messages to servers. Another major bottleneck of PowerMix is its reliance on an expensive offline protocol[2] for generating secret-shared powers of random secret field elements used to compute the powers of the actual messages.

In order to achieve the best of both worlds, we exploit the algebraic structure in powers. Our main observation is that the N_c sequential powers of field element m can be computed by evaluating a circuit of multiplication depth 1 (and algebraic degree 2) on a set of $2\sqrt{N_c} - 1$ elements that we call a square-root encoding of m. This kind of degree-2 quadratic expansion resembles protocols for private information retrieval/writing with square-root communication; however, instead of applying the expansion to sparse vectors, here we apply it to a dense vector consisting of distinct powers of a message. Employing this client message encoding results in a graceful tradeoff between server computation and client communication: servers only need $O(N_c)$ algebraic operations to compute the N_c powers of each message m and a client is now required to send the $O(\sqrt{N_c})$ secret-shares of its message m. Moreover, similarly to Blinder (and unlike PowerMix), we only require a simple

[2]In the MPC literature, an offline (sub)protocol is a protocol that is executed before the actual protocol inputs are known. See Section 2.1 for further discussion.

	t_s	Service Strength	Delivery Success	Offline Preprocess	Broadcast Required	C-to-S (Broadcast)	S-to-S (P2P)	Client (comp)	Server (comp)	Server Round
Switch-Network* [27]	$< \frac{N_s}{3}$	Fair	100%	Non-linear (MultTri)	Yes	1	$12N_sN_c\log^2 N_c$	1	$2N_cN_s\log^2 N_c$	$\log^2 N_c$
PowerMix* [27]	$< \frac{N_s}{3}$	Fair	100%	Non-linear (Powers)	Yes	1	$6{N_c}^2$	1	${N_c}^3/2$	2
Blinder [1]	$< \frac{N_s}{4}$	Robust	95%	Linear (DouRand)	Yes	$4.97\sqrt{N_c}$	$17N_c$	$4.97\sqrt{N_c}$	$5.5{N_c}^2$	4
Rabbit-Mix (This Work)	$< \frac{N_s}{4}$	Robust	$100-\varepsilon\%$	Linear (DouRand)	Yes	$1.87\sqrt{N_c}$	$5.32N_c$	$1.87\sqrt{N_c}$	$1.43{N_c}^2$	4

Table 1: Comparison to Related Works: Let N_c and N_s be the numbers of clients and servers respectively. t_s = server corruption threshold. Service strength is either Fair or Robust. The offline communication column shows the types of correlated data required to be prepared during the offline phase. DouRand, MultTri, Powers denote double random sharing, multiplication triple sharing, powers of a random sharing respectively. Offline setup marked as "linear" can be generated non-interactively using pseudorandom secret sharing. All communication complexity is measured by the number of field elements. C-to-S (broadcast) indicates communication costs from a client to each server in the number of field elements to broadcast. S-to-S (P2P) indicates the communication cost per server to be transmitted via authenticated P2P channels. Client (comp) is a client's computational overhead in N_c and N_s so that actual complexity can be computed by multiplying it by the operations required for underlying secret-sharings. Similarly, Server (comp) is also a server's computational overhead in N_c and N_s. The total communication complexity can be calculated by multiplying by N_c. * indicates that these protocols are built to work over an asynchronous network. ε denotes a statistical delivery failure percentage which is negligible (e.g., probability $\ll 1/2^{\kappa}$) for any system configurations with at least 32-byte long primes.

constant-round offline setup protocol, which can further be made entirely non-interactive in some parameter regimes.

Finally, we observe that the above quadratic compression can be viewed as using a simple *additive 2-basis* of $\mathcal{S} = \{1,2,...,N_c\}$, namely a small set $\mathcal{S}'$ such that every number in $\mathcal{S}$ can be written as a sum of two numbers from $\mathcal{S}'$. Using a more sophisticated additive 2-basis [39], we reduce the size of encoding to roughly $1.87\sqrt{N_c}$. Hence, the client communication (and computation) is smaller than $2\sqrt{N_c}$, and the server computation per client is linear in N_c.

An additional difficulty is posed by the need to guarantee robustness against malicious clients. Indeed, even a single input encoding which is inconsistent with the underlying additive 2-basis format can lead to a corruption of *all* outputs. Therefore, it is imperative for the servers to verify that the received secret-shares of an encoded client message are consistent with the additive 2-basis. We present such an efficient distributed verification protocol to verify the well-formedness of the shares contributed by the clients.

Summary of Contributions Following is a summary of the main contributions of this work.

- We construct Rabbit-Mix, a 4-round robust AB protocol with *statistically negligible* delivery failure. Rabbit-Mix has better concrete efficiency than state-of-the-art AB protocols, including those with imperfect message delivery or those that only guarantee fairness. Our protocol is built upon the existence of reliable broadcast and authenticated point-to-point channels to achieve the desired guaranteed output property, similarly to the previous related works.
- We exploit an algebraic structure in sequential powers, using an additive 2-basis to improve concrete efficiency.
- We design a distributed verification protocol to verify the well-formedness of the encoded input shares provided by clients with respect to an additive 2-basis.

Comparison to Related Works Table 1 summarizes concrete in-depth comparisons between Rabbit-Mix and relevant AB protocols with fairness or robustness. Rabbit-Mix improves over the latest robust AB protocol Blinder in all categories in terms of concrete efficiencies in both the offline and online phases. In comparison to PowerMix, Rabbit-Mix also has better concrete server communication and computation efficiencies and only requires linear correlated randomness (double random shares). This is contrasted with PowerMix and Switch-Network, which require non-linear correlated randomness that is more expensive to generate and requires interaction. In contrast, Rabbit-Mix and Blinder can both make their offline computations non-interactive by using pseudorandom secret sharing (PRSS) (See Section 7 and Appendix 2.1). On the other hand, in contrast to Switch-Network and PowerMix, Rabbit-Mix has $O(\sqrt{N_c})$ client-to-server communication overhead which might be a bottleneck in some network settings especially considering that client-to-server communication relies on reliable broadcasts. Rabbit-Mix is less tolerant against malicious servers as it has smaller corruption threshold $\frac{N_s}{4}$ than PowerMix's $\frac{N_s}{3}$. Furthermore, Rabbit-Mix has a negligible probability of delivery failure. Due to space limitations, the complexity expressions in the table only include the leading terms. We refer to Section 8 for in-depth efficiency comparisons that include empirical benchmarks.

2 Preliminaries

Let λ be the security parameter. Let a and b be integers s.t. $a \leq b$. Then, we write $[a,b]$ to denote the set of all integers $\{c \in \mathbb{Z} | a \leq c \leq b\}$ and $[a]$ to denote the set of positive integers $\{1, \cdots, a\}$. We often abuse our notations for arithmetic operations over a set. For example, let S be an ordered set of n field elements as $S = \{s_1, s_2, \cdots, s_n\}$ and a be a field element. Then, we denote the ordered set of powers of a with exponents in an ordered set S by simply a^S which is $\{a^{s_1}, a^{s_2}, \cdots, a^{s_n}\}$.

2.1 Secure Multiparty Computation

In the following, we provide the brief overview on the secure multiparty computation (MPC) especially in the honest majority setting where the number of corrupted servers is strictly less than the number of honest servers.

Shamir's Secret Sharing Shamir's Secret Sharing (SS) [34] is a core cryptographic scheme that enables one to share its secret to multiple parties where the secret remains hidden as long as a certain fraction of the parties are not compromised. SS scheme is a pair of two algorithms $(\mathsf{Share}, \mathsf{Reconst})$ defined as follows: let n be a positive integer and q be a prime such that $q > n$.

1. **Algorithm** $\mathsf{Share}_{n,d,q}(s)$: To share a secret s in a finite prime field $\mathbb{F}_q$ to n parties P_i, sharing algorithm $\mathsf{Share}_{n,d,q}(s)$ will output n secret shares denoted by ${}^{d}[\![s]\!]_q$ defined as follows: $\mathsf{Share}_{n,d,q}(s)$ will choose a random degree d polynomial $f(x)$ in $\mathbb{F}_q[x]$ with $f(0) = s$. Then, the i-th share for party P_i is $f(i)$. We omit d and q as well when the context is clear. The computational cost is $O(d \log^2 d)$.

2. **Algorithm** $\mathsf{Reconst}_{n,d,q}([\![s]\!])$: To reconstruct a secret s, we need to interpolate a d-degree polynomial $f(x)$ such that $f(0) = s$ from ${}^{d}[\![s]\!]_q$ which is n distinct coordinates $\{(i, f(i))\}_{i \in [n]}$ with up to t errors. We abuse the notation and use $\mathsf{Reconst}_{n,d,q}([\![s]\!])$ for the sake of simpler notation when the context is clear. We use Reed-Solomon (RS) decoding algorithm to implement $\mathsf{Reconst}_{n,d,q}([\![s]\!])$ which will interpolate a polynomial $f(x)$ of degree at most d from n points correcting up to $t = \lfloor (n-d)/2 \rfloor$ errors. The most efficient known version of the decoding algorithm is based on FFTs which has $O(n \log^2 n)$ computation [36].

The d-sharing of secret s, ${}^{d}[\![s]\!]$ is said to be *d-consistent* if there exists a polynomial $f(x)$ of at most degree d such that every honest party P_i holds share $f(i)$.

Batched Robust Opening In order to publicly open secrets, we use a robust share opening protocol due to the works [4,15] which opens multiple secret sharings in a batched manner. The idea is to expand ℓ consistent d-sharings into N consistent d-sharings for $\ell = N - (2t+1)$ and $d \leq 2t$ by using a public Vandermonde matrix. Let $Van^{(N \times \ell)}$ (simply Van when the context is clear) be a Vandermonde matrix where the entry at the i-th row and j-th column of the matrix is i^j. Then to open ${}^{d}[\![x^{(1)}]\!]_q, \cdots, {}^{d}[\![x^{(\ell)}]\!]_q$, all parties locally compute the matrix multiplication of Van with the vector of $[\![x_1]\!], \cdots, [\![x_\ell]\!]$, which essentially computes consistent d-shares of secret degree-ℓ polynomial with shared coefficients x_1, x_2, $\cdots$, x_ℓ. Then, all parties will reconstruct at least $(N-t)$ y_i's which can be consequently reconstructed to x_i's. We denote the batched robust opening protocol by Π_{open}^{robust} that takes inputs ℓ *d-consistent* sharings ${}^{d}[\![x^{(1)}]\!]_q, \cdots, {}^{d}[\![x^{(\ell)}]\!]_q$ and outputs $x^{(1)}, \cdots, x^{(\ell)}$.

Protocol 1: Robust Opening Protocol Π_{open}^{robust}

Parameters: N is the number of parties (servers), t is the corruption threshold.
Input: ℓ *d-consistent* secret-sharings ${}^{d}[\![x^{(1)}]\!]_q, \cdots, {}^{d}[\![x^{(\ell)}]\!]_q$.
Output: $x^{(1)}, \cdots, x^{(\ell)}$.

1: Each party P_i locally computes $Van^{N \times \ell}({}^{d}[\![x^{(1)}]\!]_q, \cdots, {}^{d}[\![x^{(\ell)}]\!]_q) \rightarrow {}^{d}[\![y^{(1)}]\!]_q, \cdots, {}^{d}[\![x^{(N)}]\!]_q$
2: Each party sends share ${}^{d}[\![y^{(i)}]\!]_q$ to party P_i for each $i \in [N]$.
3: Each party P_i reconstructs $y^{(i)}$ by running $\mathsf{Reconst}_{N,d,q}(\left\{{}^{d}[\![y^{(i)}]\!]_q\right\}_{i \in [N]})$.
4: Each party P_i sends $y^{(i)}$ to other parties.
5: Each party upon all $y^{(i)}$'s runs $\mathsf{Reconst}_{N,d,q}(\left\{y^{(i)}\right\}_{i \in [N]})$ that outputs a degree-$(\ell-1)$ polynomial with coefficients $x^{(1)}, \cdots, x^{(\ell)}$.

The concrete communication cost to open $\ell = (N_s - t_s)$ secret-shares is $2N_s$ elements by each server as $t_s < N_s/4$ in this work. The computational cost by each server is $\frac{3}{4}{N_s}^2$ field multiplications/additions and two instances of $\mathsf{Reconst}_{N_s,d,q}$.

Algebraic MPC over Shamir's Shares We use a MPC protocol of basic arithmetic operations secure against malicious adversary with corruption threshold $t_s < N_s/4$ where a point-to-point authenticated communication and a broadcast channels are available to the MPC servers [3,4,15]. Let x and y be two secret input values and z be a public input values over a prime field $\mathbb{F}_q$. Then, the following secure arithmetic computations will be used to build our anonymous broadcast protocol:

1. **Share addition:** Each party holds input secret-shares ${}^{d}[\![x]\!]_q$ and ${}^{d}[\![y]\!]_q$, then locally computes ${}^{d}[\![x]\!] + {}^{d}[\![y]\!] = {}^{d}[\![x+y]\!]$

2. **Share multiplication**: Each party holds input secret-shares ${}^{d}[\![x]\!]$ and ${}^{d}[\![y]\!]$, then locally computes ${}^{d}[\![x]\!] \cdot {}^{d}[\![y]\!] = {}^{2d}[\![x \cdot y]\!]$.

3. **Scalar multiplication:** Each party holds input secret-shares ${}^{d}[\![x]\!]$ and a publicly known scalar z, then locally computes ${}^{d}[\![x]\!] \cdot z = {}^{d}[\![x \cdot z]\!]$.

4. **Scalar addition:** Each party holds input secret-shares ${}^{d}[\![x]\!]$ and a publicly known scalar z, then locally computes ${}^{d}[\![x]\!] + z = {}^{d}[\![x+z]\!]$.

In this work, we design our algebraic circuits to only have the maximum multiplication depth 1. Therefore, secure multiplications over two d-shares ${}^{d}[\![x]\!]_q$ and ${}^{d}[\![y]\!]_q$ in our work will be simply local multiplications as ${}^{2d}[\![xy]\!]_q = {}^{d}[\![x]\!]_q {}^{d}[\![y]\!]_q$ resulting in a $2d$-share. Note that xy cannot be reconstructed by directly applying $\mathsf{Reconst}_{n,d,q}$ on the $2d$-shares due to non-trivial information being leaked about secrets x and y. Thus, we rerandomize the $2d$-shares output by local multiplications of two shares by employing preprocessed random double sharing [15] (see Section 2.1). A random double sharing is a pair of ${}^{d}[\![r]\!]_q$ and ${}^{2d}[\![R]\!]_q$ such that r is a uniformly random secret over $\mathbb{F}_q$ and $r = R$. With ${}^{d}[\![r]\!]_q$ and ${}^{2d}[\![R]\!]_q$, we rerandomize ${}^{2d}[\![xy]\!]_q$ by computing ${}^{2d}[\![xy]\!]_q + {}^{2d}[\![R]\!]_q - {}^{d}[\![r]\!]_q$.

Security of Algebraic MPC In this work, we rely on standard MPC components and protocols from secure MPC protocols [3, 4, 15, 20] that securely realizes necessary MPC functionalities such as additions, multiplications, coin flip, etc., in the honest-majority setting. Specifically, we use the MPC components to construct our AB protocol by designing algebraic circuits of our AB functionality and composing the standard MPC protocols to securely evaluate the algebraic circuits. The protocol is secure as long as the number of corrupted servers $t < \frac{N_s}{4}$ where the arbitrary composition of protocols preserves the security [4, 9, 15].We describe our protocols in the hybrid model following [9] where the security of protocol Π that runs a subprotocol Π_{sub} is guaranteed even if Π_{sub} is replaced with its ideal functionality $\mathcal{F}_{sub}$. In such case, we say that Π is a secure protocol in the $\mathcal{F}_{sub}$-hybrid model.

Coin Flip We use a coin flip functionality denoted by $\mathcal{F}_{coin}$ when invoked with parameters a prime number q and $\ell \in \mathbb{Z}^+$ produces a set of ℓ field elements uniformly sampled random from $\mathbb{F}_q$ to participating servers. We can realize the functionality with information theoretic security by using a random sharing protocol (described later in this section) where all parties preprocess ℓ random shares over $\mathbb{F}_q$ and open them when the coins are necessary.

Offline Functionalities/Protocols Our AB protocol executes standard offline MPC protocols to prepare (correlated) randomness information that it uses during the execution of the main MPC protocols. In the MPC literature, we call these protocols/functionalities *offline* computation/functionality so that MPC servers before getting their inputs execute these protocols to prepare randomness or correlated information to execute the main MPC protocol upon their inputs. We use the following offline functionalities and see [15] for more details:

1. **Random Sharing:** A random sharing functionality denoted by $\mathcal{F}_{rand}$ takes N, degree d, and $\ell \in \mathbb{Z}^+$ and outputs consistent d-sharings of ℓ random field elements from $\mathbb{F}_q$.

2. **Random Double Sharing:** A random double sharing functionality denoted by $\mathcal{F}_{DouRan}()$ takes parameters N, degree d, and $\ell \in \mathbb{Z}^+$ and outputs ℓ pairs of $({}^{d}[\![r_i]\!], {}^{2d}[\![R_i]\!])$ such that random $r_i = R_i$ for all $i \in [\ell]$.

We measure the offline efficiency of a protocol by the number of random double sharings required to execute its online phase protocol.

2.2 Anonymous Broadcast

An anonymous broadcast (AB) functionality is involved with N_s servers, denoted by $P_1, P_2, \cdots, P_{N_s}$, which jointly serve at maximum N_c clients[3], in an epoch such that $N_s \ll N_c$. An adversary $\mathcal{A}$ may corrupt at most t_s servers as well as at most t_c clients where we require $t_s < N_s/c$ for some constant at least 2 while $t_c < N_c$. Given these, we define the ideal robust AB functionality as follows.

Definition 1 (Robust AB Functionality $\mathcal{F}_{AB}$)**.** *Let $C_1, C_2, \cdots, C_{N_c}$ be participating clients where C_i has secret message m_i such that an adversary $\mathcal{A}$ controls at most t_c clients for some $t_c < N_c$. Let a client message be represented as a field element over a prime field $\mathbb{F}_q$ for some prime q. Then, the ideal AB functionality denoted by $\mathcal{F}_{AB}$ proceeds as follows:*

1. *Functionality $\mathcal{F}_{AB}$ initialize an empty set H and waits for a message $m_i \in \mathbb{F}_q$ from each client.*

2. *An adversary $\mathcal{A}$ may select and corrupt up to t_c clients and instruct these clients to send any message from $\mathbb{F}_q \cup \{\perp\}$ where $\perp$ is a special character indicating an invalid input message.*

3. *Upon each m_i's, $\mathcal{F}_{AB}$ checks if $m_i = \perp$. If so, drop it. Otherwise, add m_i to H.*

4. *Once all messages are received from all participating clients, $\mathcal{F}_{AB}$ shuffles m_i's in H and outputs the messages in a random order.*

We note that the above definition of functionality formalizes the *robust* anonymous broadcast functionality in the presence of malicious adversary. That is, regardless of the malicious activities orchestrated by an adversary $\mathcal{A}$, the functionality $\mathcal{F}_{AB}$ is guaranteed to receive, mix, and publish messages from all honestly behaving clients. The fair AB functionality will allow an adversary to terminate the protocol execution without delivering the messages to clients. Hence, the ideal

[3] In general, N_c is a public parameter that sets only the maximum number of messages per epoch, while not limiting the total number of clients.

functionality for fair AB can be defined by modifying Step 4 in the above definition as follows: Upon completion of shuffling messages in H, $\mathcal{F}_{AB}$ asks if $\mathcal{A}$ wishes to obtain the messages. Finally, if $\mathcal{A}$ responses with $\perp$, then $\mathcal{F}_{AB}$ outputs $\perp$ to all clients and $\mathcal{A}$. Otherwise, $\mathcal{F}_{AB}$ outputs all the messages to clients as well as $\mathcal{A}$. Finally, unfair AB functionality allows an adversary to obtain all messages from honest clients and to determine which messages from honest clients are published. For this definition, Step 4 is modified as follows: Upon completion of shuffling messages in H, $\mathcal{F}_{AB}$ sends all the shuffled messages to $\mathcal{A}$. Then, $\mathcal{A}$ returns to $\mathcal{F}_{AB}$ a subset of messages denoted by H'. $\mathcal{F}_{AB}$ then publishes messages in H'.

3 Warm-up Protocol

To provide the intuition behind the mixing protocol, we start with the most basic mixing protocol where all servers and clients are assumed to follow the protocol. The toy protocol can be seen as a variant of PowerMix [27] where the server computation and c-to-s communication are re-balanced so that server computation is minimized at the cost of high c-to-s communication.

Algebraic Mixing Protocol Consider that clients C_i want to send a message $m_i \in \mathbb{F}_q$. The algebraic mixing protocol is based on Newton's identities. The mixing protocol can be summarized as a three-step process. First, suppose that mixing committee servers have the shares of the j-th powers of each $[\![m_i]\!]$ for each $j \in [N_c]$, that is, $[\![{m_i}^1]\!]$, $[\![{m_i}^2]\!]$, $\cdots$, $[\![{m_i}^{N_c}]\!]$. Then, the servers jointly compute power sums $x_j = {m_1}^j + {m_2}^j + \cdots + {m_{N_c}}^j$ for each power j (i.e., by locally summing the shares of the same power and reconstructing the sums). Note that computing power sums is an essential step to achieve desired mixing property. This step converts all message power encodings into order-independent aggregate values due to commutativity of additions. The second step is to locally compute a degree-N_c polynomial $f(x) = x^{N_c} + c_{N_c-1}x^{N_c-1} + \cdots + c_0$ such that $f(x)$'s roots are m_1, m_2, $\cdots$ m_{N_c} by using Newton's identities [29, 41] on the power sums to recursively compute the coefficients as follows:

$$\begin{aligned}
c_{N_c-1} &= -x_1 \\
c_{N_c-2} &= -(x_2 + c_{N_c-1}x_1)/2, \\
c_{N_c-3} &= -(x_3 + c_{N_c-1}x_2 + c_{N_c-2}x_1)/3, \\
&\cdots
\end{aligned}$$

We denote this procedure by $\mathsf{NewtonId}(x_1, x_2, \cdots, x_{N_c})$. The computation of $\mathsf{NewtonId}$ takes ${N_c}^2/2$ additions and multiplications. See [29] for more details on Newton's identities.

Finally, the servers locally executes a root-finding algorithm $\mathsf{RootFind}$ on $f(x)$ which eventually outputs roots of $f(x)$ which are m_1, m_2, $\cdots$, m_{N_c} in a random order.

We instantiate $\mathsf{RootFind}$ with a randomized algorithm by Grenet, Hoeven, and Lecerf (algorithm 11 in [23]) which finds roots of polynomial $f(x)$ of degree d over a prime field $\mathbb{F}_q$ with prime $q = M \cdot 2^m + 1$ where $d < q$ and $M = O(\log q)$, and 2^m is chosen to be much larger than the number of terms to be found. The average running time of the root finding algorithm is $O(d(\log^3 q \log d + \log^4 q))$.

Naive Client Message Encoding Suppose that a client wishes to broadcast message m, a field element in $\mathbb{F}_q$. A client first needs to encode m into an AM-friendly format. With N_c being the number of participating clients in an epoch, a naive AM-friendly encoding $\mathsf{E}^0 : \mathbb{F}_q \to (\mathbb{F}_q)^{N_c}$ is simply the ordered set $\mathsf{E}^0(m) = \mathsf{pkg}_m = \{m^1, m^2, \cdots, m^{N_c}\}$.

Warm-up Protocol Let N_c and N_s be the numbers of clients and servers respectively. First, each client with message m locally computes $\mathsf{E}^0(m)$, secret-shares pkg_m, and send the shares $[\![\mathsf{pkg}_m]\!]$ to appropriate servers. Each server wait for all client message submissions and then locally compute the shared sums of powers described above over secret-shares. Finally, servers will jointly open the sums of powers and locally computes a symmetric polynomial and run $\mathsf{RootFind}$ to output mixed client messages. As long as all clients and servers honestly follow the protocol, this warm-up protocol is immediately correct. The protocol is secure as only the sums of powers are opened whereas all individual powers of messages remain shared.

4 Compressing C-to-S Communication

In this section, we describe how to optimize the warm-up protocol so that client's computation and c-to-s communication per server are sub-linear in the number of clients.

The First Attempt We observe that a sequence of N_c powers of a field element can be encoded into a set of $2\sqrt{N_c} - 1$ field elements in a way that the original N_c powers can be recovered (decompressed) with multiplications of two elements from the encoding. In the further details, consider that N_c is a square of some integer and a client has a message m. Then, the client can encode m into $\left\{m^1, m^2, \cdots, m^{\sqrt{N_c}-1}, m^{\sqrt{N_c}}, m^{2\sqrt{N_c}}, \cdots, m^{(\sqrt{N_c}-1)\sqrt{N_c}}\right\}$ instead of the entire sequence of N_c powers. Given this subset of powers of m, it is easy to see that multiplications of two entries from the subset will give $\{m^1, m^2, \cdots, m^{N_c}\}$ where we call this computation *an input decompression*. The size of the naive encoding is $2\sqrt{N_c} - 1$.

Additive 2-basis Above, we observe that multiplications of two encodings to enumerate all N_c powers are essentially additions of two entries' exponents to enumerate all exponents

from 1 to N_c. In mathematics, the later is known as an additive 2-basis for set $[N_c]$. We adopt the classical optimization of the additive 2-basis due to Mrose [39]. We denote Mrose's additive 2-basis for a set $[N_c]$ by $\mathcal{B}_{N_c}$ defined as follows. Let N be $N = 14L^2 + 10L - 1$, then additive 2-bases $\mathcal{B}$ for set $[N]$ contains five distinct finite arithmetic progressions:

$$S_1 = \{1 + (i-1)1 \mid \forall i \in \mathbb{Z} \text{ s.t. } 1 \leq i \leq L\},$$
$$S_2 = \{2L + (i-1)L \mid \forall i \in \mathbb{Z} \text{ s.t. } 1 \leq i \leq 3L\},$$
$$S_3 = \left\{3L^2 + 2L + (i-1)(L+1) \mid \forall i \in \mathbb{Z} \text{ s.t. } 1 \leq i \leq L\right\},$$
$$S_4 = \left\{6L^2 + 4L + (i-1)1 \mid \forall i \in \mathbb{Z} \text{ s.t. } 1 \leq i \leq L+1\right\},$$
$$S_5 = \left\{10L^2 + 7L + (i-1)1 \mid \forall i \in \mathbb{Z} \text{ s.t. } 1 \leq i \leq L+1\right\}.$$

The sizes of arithmetic progressions are: $|S_1| = |S_3| = L$, $|S_2| = 3L$, and $|S_4| = |S_5| = L+1$ so that $|\mathcal{B}| = 7L+2$ in total. Approximately, $7N \approx 2|\mathcal{B}|^2$, so that the size of the additive 2-basis $|\mathcal{B}| \approx \sqrt{\frac{7N}{2}}$ which leads to $|\mathcal{B}| \approx 1.87\sqrt{N}$ and $L \approx 0.267\sqrt{N}$. See [32] and [31] for further details and another variant with the same asymptotic properties.

Decompressing $\mathcal{B}$ to set $[N]$ can be done by adding two entries from the basis in a certain way. Due to space restriction, we provide the additive 2-basis decompression circuit $\mathsf{Decomp}(\cdot)$ in Circuit 1 in Appendix A.

Client's Input Encoding/Submission Now we describe how a client with message $m \in \mathbb{F}_q$ submits its messages to servers. Let $N_c = 14L^2 + 10L - 1$ be *the maximum number* of participating clients in an epoch for some integer L.[4] Given the corresponding (precomputed) additive 2-basis $\mathcal{B}$, a client with message m simply encodes m by computing $m^{\mathcal{B}}$, denoted by pkg. Then, the client computes the secret-sharing ${}^{t_s}[\![\mathsf{pkg}]\!]_q$ by running $\mathsf{Share}_{N_s,t_s,q}(x_i)$ for each $x_i \in \mathsf{pkg}$ and send ${}^{t_s}[\![\mathsf{pkg}]\!]_q$ to servers accordingly. We denote this client input encoding submission protocol by Π_{submit} that takes a client input message m and servers outputs their shares of ${}^{t_s}[\![\mathsf{pkg}]\!]_q$ at the end of protocol.

Protocol 2: Client Message Submission Protocol Π_{submit}

Parameters: Prime q, the number of clients N_c, the number of servers N_s, the server corruption threshold t_s, and the additive 2-basis $\mathcal{B}$.
Input: Client C holds a message m defined over $\mathbb{F}_q$.
Output: Servers output the secret-shares of ${}^{t_s}[\![\mathsf{pkg}]\!]_q$.

1: C computes $\mathsf{pkg}_m \leftarrow m^{\mathcal{B}}$.
2: C computes ${}^{t_s}[\![\mathsf{pkg}]\!]_q \leftarrow \mathsf{Share}_{N_s,t_s,q}(\mathsf{pkg}_m)$.
3: C sends the i-share of ${}^{t_s}[\![\mathsf{pkg}]\!]_q$ to S_i.

Each client's communication complexity is $1.87\sqrt{N_c}$ field elements per server. The computational complexity is $1.87\sqrt{N_c}$ multiplications followed by the computational cost from $1.87\sqrt{N_c}$ instances of Shamir's secret-sharing. We note that both complexities of computing and transmitting a client message encoding based on the additive 2-basis offers roughly 10% improvement compared to the ones based on the first attempt encoding described above where the message encoding is of size $2\sqrt{N_c} - 1$.

Decompressing Client Input Shares Upon clients' shared input message encodings ${}^{d}[\![m^{\mathcal{B}}]\!]_q$, the servers locally decompress them to the secret-shares of $m^{[N_c]}$. Having the additive 2-basis $\mathcal{B}$ as exponents in the message encoding, the encoding decompression to shares of $m^{[N_c]}$ can be done by locally computing ${}^{2d}[\![m^i]\!]_q = [\![m^i]\!] \cdot [\![m^j]\!]$ from ${}^{d}[\![m^{\mathcal{B}}]\!]_q$, whenever the additive 2-basis decompression requires the addition of i and j from B. Note that the resulting output shares are the $2d$-secret-shares of N_c powers of message m that has degree at most $2d$ as the maximum multiplication depth is 1. We denote this decompression protocol by $\Pi_{Decomp}({}^{d}[\![m^{\mathcal{B}}]\!]_q)$ that takes as inputs secret-shares ${}^{d}[\![m^{\mathcal{B}}]\!]_q$ and outputs ${}^{2d}[\![m^{[N_c]}]\!]_q$. See Protocol 4 for the full description in Appendix B.

The correctness immediately follows from the property of additive 2-basis decompression described in Circuit 1 as the decompression here is carried out on the exponents via the secure multiplications. The computational costs of the decompression protocol is essentially $14L^2 + 9L \approx N_c - 0.267\sqrt{N_c}$ field multiplications.

5 Input Well-formedness Verification

Malicious clients may deviate in two distinct ways from the protocol instruction submitting shares of malformed message packages. The first deviation strategy is that a malicious client submits a t-consistent shares of an input message encoding but the encoding does not agree with (is not well-formed w.r.t.) the additive 2-basis format. The second is that a malicious client submits inconsistent shares of an input message encoding. In both cases, the AB protocol will fail to output messages. To defeat such DoS attacks, servers must ensure that input message packages are (1) well-formed according to the additive 2-basis for N_c and (2) t-consistent. In the section, we present an efficient input encoding verification protocol by devising an efficient distributed verification protocol to defeat the first adversarial strategy. Looking ahead, we will show how to defeat the second strategy in Section 7.

Verifiable Input Encoding Let $N_c = 14L^2 + 10L - 1$, $\mathcal{B} = \{S_1, S_2, \cdots, S_5\}$ be the additive 2-basis for N_c described in the previous section and finally m be a client message. Once a client completes the submission of message m, servers holds the secret-sharings of input package $\mathsf{pkg} = \{x_1, x_2, \cdots, x_{7L+2}\}$. Let sets X_1, X_2, $\cdots$, X_5 be the 5 partitions of pkg where each X_i corresponds to m^{S_i} with denoting the

[4] We remind that N_c is a public system parameter that upper-bounds the number of participating clients in an epoch and *does not require* the number of clients to be N_c in order for the AB protocol to work. That is, the Rabbit-Mix's functionality works correctly even if the actual participating clients is smaller than N_c in an epoch.

elements of set X_i by $x_{i,1}, x_{i,2}, \cdots$. Then, servers must jointly verify that $\mathsf{pkg} = \{m^{\mathcal{B}}\}$ for some m. This essentially means that the servers must be able to verify that the linear relations of the arithmetic progressions $S_1, S_2, \cdots, S_5$ appear as exponents of m in the fixed order. We have two distinct types of conditions to verify:

- **(Inner-Progression)** This type, called inner-progression of X_i is about whether each set X_i have an arithmetic progression of S_i as exponents. We define b_i^0 to be a linear sketch which is 0 only if the inner progression of X_i is satisfied. If an input package pkg is well-formed according to $\mathcal{B}$, then all b_i^0's will be 0.
- **(Inter-Progression)** This condition type, called inter-progression of X_i for $i \in \{2, \cdots, 5\}$ is about whether X_i's first element is correctly formed in relation to two elements in X_j for $j < i$. We define b_i^1 to be a linear sketch which is 0 only if the first element of X_i is indeed the first element of m^{S_i}.

In order to verify these conditions, we essentially exploit an system of degree-2 verification to which a client message encoding of additive 2-basis is a unique solution.

Inner Progression Verification We outline how to compute linear sketches for inner-progression conditions. Recall $S_1 = \{1, 2, \cdots, L\}$ starting from 1. This implies $x_{1,1} = m$. Then, we can define linear sketch $b_1^0 = \sum_{i=1}^{L-1} c_i(x_{1,i+1} - x_{1,i} \cdot x_{1,1})$ for random coin c_i's such that $b_1^0 = 0$ if the inner condition holds for X_1. The random coin c_i is distributed uniformly and must be multiplied to $x_{1,i+1} - x_{1,i} \cdot x_{1,1}$ to prevent an adversary from maliciously choosing S_1 such that $0 = \sum_{i=1}^{L-1}(x_{1,i+1} - x_{1,i} \cdot x_{1,1})$. Linear sketches b_2^0, b_4^0, and b_5^0 for X_2, X_4, and X_5 can be similarly defined as b_1^0 relying on the existing entries in pkg.

Defining b_3^0 for X_3 requires adding an additional element to the client's input encoding for the verification purpose. S_3 is an arithmetic progression of difference $L+1$ which does not exist in S_1 nor S_2 and can only be computed from the addition of 1 and L from S_1 which creates an additional multiplication depth for the verification. We bypass this by introducing a new ordered set denoted by S_6 that contains $L+1$. Then, the input submission package of message m has additionally X_6 with $x_{6,1} = {x_{1,1}}^{L+1}$ which is supposedly m^{L+1} for message m. Given $x_{6,1}$, the linear sketch of inner progression for X_3 is defined as $b_3^0 = \sum_{i=1}^{L-1} c_i(x_{3,i+1} - x_{3,i} \cdot x_{6,1}) + c(x_{6,1} - x_{1,1} \cdot x_{1,L})$ for random coins c_i and c where the computation still has multiplication depth 1. Note that $b_3^0 = 0$ only if the inner progression condition for X_3 is satisfied and $x_{6,1}$ is ${x_{1,1}}^{L+1}$.

Inter Progression Verification Except for X_1, each X_i's first element needs to be verified to be correct. If the first element of X_i is incorrect, then it is possible for X_i to still satisfy its inner progression while not matching with the correct additive 2-basis. Specifically, for each $i \in \{2, \cdots, 5\}$, we want to ensure $x_{i,1} = {x_{1,1}}^{s_{i,1}}$.

The linear sketch of inter progression for X_2 can be simply computed as $b_2^1 = c(x_{1,L} \cdot x_{1,L} - x_{2,1})$ for fresh random coin c since $s_{2,1} = 2L$ and $s_{1,L} = L$. The inter progression linear sketch for X_3 is also simple to compute as $b_3^1 = c(x_{2,3L} \cdot x_{1,L} - x_{3,1})$ for fresh random coin c. Also, the inter progression linear sketch for X_4 is simply computed as for fresh random coin c, $b_4^1 = c(x_{4,1} - x_{3,1} \cdot x_{3,1})$.

Verifying inter progression for X_5 requires a special treatment. S_5's first element is $10L^2 + 7L$ that cannot be expressed as a product of two elements in X_1, X_2, X_3, or X_4. Therefore, to keep multiplication depth 1 in general, we introduce a new exponent $s_{6,2}$ into S_6 such that (1) $s_{6,2}$ is an addition of two elements in any of S_j for $j < 5$ and (2) the exponent of X_5's first element (which is $10L^2 + 7L$) is an addition of $x_{6,2}$ and another element x in any of S_j for $j < 5$. We choose $s_{6,2}$ to be $10L^2 + 7L - 1$ additionally introduced into S_6 which is (1) an addition of $s_{3,L} = 4L^2 + 2L - 1$ and $s_{4,L+1} = 6L^2 + 5L$ from X_3 and X_4 respectively and (2) satisfies that $s_{5,1}$ is an addition of $s_{6,2} + s_{1,1}$. Hence, the linear sketch of inter progression for X_5 is defined as $b_5^1 = c_1(x_{5,1} - x_{6,3} \cdot x_{1,1}) + c_2(x_{6,3} - x_{3,L} \cdot x_{4,L+1})$ for fresh random coins c_1 and c_2.

In fact, the additive 2-basis based client message encoding by itself is verifiable without requiring introduction of S_6. We defer this to the full version of the paper.

Verifiable Input Encoding/Submission Due to space restriction, we provide the full description of the client input submission protocol with well-formedness verfiability (Protocol 5) in Appendix D. The new protocol is identical to the encoding/submission protocol described in Section 4 except adding $S_6 = \{L+1, 10L^2 + 7L - 1\}$ to additive 2-basis $\mathcal{B}$. Thus, with two additional field elements, the c-to-s communication cost is $7L + 4$ field elements per server which is still approximately $1.87\sqrt{N_c}$. The new input submission protocol $\Pi_{Submit}(m)$ will have client with input message m computes and secret-shares m^{S_i} for $i \in [6]$ and finally send the shares to servers accordingly.

Input Well-formedness Verification Protocol For the well-formedness verification, servers proceed as follows. Once they receive a secret-shared client input package pkg, they run coin-flip protocol $\mathcal{F}_{coin}$ to generate $1.87\sqrt{N_c}$ random coins. Then, each server locally computes the secret-share of sum of all linear sketches ${}^{2t}[\![b]\!]_q = \sum_{i=0}^{1} \sum_{j=1}^{5} {}^{2t}[\![b_j^i]\!]_q$. Finally, servers jointly reconstruct and check b. If $b = 0$, servers will keep pkg, otherwise delete it. Note that it is possible that $b = 0$ with probability $1/|\mathbb{F}|$ while the input is malformed. Let κ be the statistical soundness parameter. The above verification needs to be executed $\kappa/\log q$ times with independent input challenge c_i's to achieve the desired soundness error probability $1/2^{\kappa}$. We denote the well-formedness verification protocol by $\Pi_{ver}()$ and see Protocol 6 in Appendix E.

6 Rabbit-Mix against Malicious Adversaries

We describe our secure Rabbit-Mix protocol secure against malicious clients and servers without fairness. Looking ahead, we transform this unfair AB protocol into a robust protocol with tools for robustness in Section 7. We put together all the pieces built so far into an anonymous broadcast protocol that securely realizes the functionality $\mathcal{F}_{AB}$ according to Definition 1. The complexity analysis will be provided in Section 8.

Protocol 3: Anonymous Broadcast Π_{AB}

Input Parameters: Let q be a prime. Let $N_c = 14L^2 + 10L - 1$ be the number of participating clients in the current epoch. Let N_s be the number of servers.

Preprocessed Information: $2N_c$ *consistent* random double sharings $({}^{t}[\![r_i]\!], {}^{2t}[\![R_i]\!])$ for $i \in [N_c]$ prepared by running $\mathcal{F}_{DouRan}(N_c)$ in the offline phase. In addition, μ sets of $(7L+3)$ random coins each denoted by $\left\{c^{(i)}\right\}_{j\in[7L+3]}$ prepared by invoking $\mathcal{F}_{coin}(\mu(7L+4))$ such that $\mu = \kappa/\log q$.

Input: All servers receive and hold message package shares ${}^{t}[\![\mathsf{pkg}^{(i)}]\!]_q$ from all clients C_i via Π_{submit} (Protocol 5 in Appendix D). A set of identified malicious clients $\mathcal{C}^*$ initialized to be an empty set.

Output: The servers output $m_1, m_2, \cdots, m_{N_c}$ in a randomly shuffled order.

1: **(Format Verification)**: repeat μ times with random coins.
2: Each server locally computes for each $i \in [N_c]$, ${}^{2t}[\![b^{(i)}]\!]_q \leftarrow \Pi_{ver}([\![\mathsf{pkg}^{(i)}]\!], \left\{c^{(i)}\right\})$.
3: All servers open all $b^{(i)}$'s as $\Pi^{robust}_{open}(\left\{{}^{2t}[\![b^{(i)}]\!]_q\right\})$.
4: Each server updates set $\mathcal{C}^*$: for every $i \in [N_c]$, do the followings:
   1. If any opening returns $\perp$, output $\perp$ and terminate.
   2. If $b^{(i)}$ is a non-zero value, add the client to $\mathcal{C}^*$.
5: Set $\mathcal{C}^H = \mathcal{C} \setminus \mathcal{C}^*$, a set of clients' indexes that submitted inputs of valid format.
6: **(Input Decompression and Mixing)**
7: Each server locally runs for each $i \in \mathcal{C}^H$, $\Pi_{Decomp}([\![\mathsf{pkg}^{(i)}]\!])$ to decompress the input shares into $\{[\![{m_i}^1]\!], [\![{m_i}^2]\!], \cdots, [\![{m_i}^{N_c}]\!]\}$.
8: Each server locally computes for each $j \in [N_c]$, $[\![x_j]\!] = \sum_{i\in\mathcal{C}^H} [\![{m_i}^j]\!]$.
9: All servers open all x_j's by running $\Pi^{robust}_{open}(\{[\![x_j]\!]\})$.
10: Locally apply Newton's identity to compute a symmetric polynomial $f(x)$ of degree N_c using x_j's.
11: Run the root finding algorithm $\mathsf{RootFind}$ on $f(x)$ which returns the set of $\{m_i\}_{i\in\mathcal{C}^H}$ in a shuffled order which is the protocol output.

Theorem 1. *For any malicious adversary corrupting some constant $t_c < N_c$ clients and $t_s < N_s/4$ servers and for any client messages m_1, m_2,$\cdots$,m_{N_c}, protocol Π_{AB} securely and statistically realizes unfair $\mathcal{F}_{AB}$ with anonymity set size at least $N_c - t_c$ as defined in Definition 1.*

Proof. Intuitively, our simulator (ideal world adversary) $\mathcal{S}$ simulates the view of adversary in the real execution with an access to the ideal functionality $\mathcal{F}_{AB}$. Let $\mathcal{A}$ be an adversary corrupting less than t_s servers. $\mathcal{S}$ starts simulating honest servers' computation by initiating C^* to be an empty set and invoking $\mathcal{F}_{AB}$. $\mathcal{S}$ locally computing verification circuit on corrupted clients' input shares given by $\mathcal{A}$. Then, $\mathcal{S}$ opens every verification bit b_i for the i-th submitted input message. If any reconstructions fail, $\mathcal{S}$ sends a termination signal to $\mathcal{F}_{AB}$ and ends the simulation. Otherwise, $\mathcal{S}$ updates C^* with i's such that $b_i \neq 0$. $\mathcal{S}$ simulates the rest: it obtains all corrupted clients' inputs $\{m_i\}$ such that $m_i = \perp$ for $i \in C^*$. $\mathcal{S}$ sends corrupted clients inputs to functionality $\mathcal{F}_{AB}$ which returns all messages $\{m_i\}$ including honest clients' messages. $\mathcal{S}$ simulates the rest of interaction to provide $\mathcal{A}$ with $\{m_i\}$ by simulating Π^{robust}_{open} where $\mathcal{S}$ possibly obtains a subset of $\{m_i\}$ as its output. Finally, $\mathcal{S}$ submits the returned subset of $\{m_i\}$ to $\mathbb{F}_{AB}$ and completes the simulation. The view of adversary in the real execution is identical to the simulated view in the ideal execution except with probability $t_c/2^\kappa$ where κ is a security parameter since two views are different when $\mathcal{A}$ in the real execution succeeds to pass input well-formedness verification with malformed message packages from corrupted clients. We note that the sender-anonymity is guaranteed regardless of adversary's strategy since messages from honest clients are reconstructed only at the end of protocol after the algebraic mixing is correctly completed. □

Handling Large Messages The above description assumes that the maximum message length of clients is upper-bounded by $\log|\mathbb{F}|$ bits. For larger messages, we let clients and servers do as follows: now clients and servers have an additional parameter for the maximum number of sub-messages per epoch, denoted by μ. Clients then partition their messages into μ sub-messages each attached with the same random tag τ of fixed length (e.g., statistical parameter $\kappa \geq 40$) so that m is partitioned into $m_1||\tau$, $\cdots$, $m_\mu||\tau$ where each $m_i||\tau$ can be encoded in $\mathbb{F}$. Then clients submit each sub-message to mixing servers. To avoid deanonymization by different message lengths, clients with shorter messages must pad their messages. Servers wait for all clients completing the submissions of all μ sub-messages (one sub-message for each AB protocol instance). Once all submissions are completed, servers execute the μ instances of AB protocol. Since servers don't execute AB protocols until all sub-messages arrive, corrupted servers/clients cannot corrupt messages with identical tags except with very small chance.

7 Rabbit-Mix with Robustness

We transform the AB protocol in Section 6 to provide the robustness as described in Section 2.2. The main technical problem is that bad shares can be dealt by malicious clients or

servers so that batch reconstruction fails to reconstruct secrets resulting in the abort of protocol execution. This means there are two places that should be upgraded for Protocol 3 to be robust according to Definition 1: (1) client message input submission protocol and (2) random double sharing protocol (including random sharing protocol). Obviously, using a verifiable secret sharing (VSS) protocol, for example [6], to deal every secret for both protocols will immediately resolve the entire technical issue, resulting in a robust AB protocol. However, VSS protocols are expensive, relying on the interactions of multiple rounds or other heavy cryptography tools. Hence, we rely on the combination of dispute control techniques as well as lightweight robust input protocol by [5, 15] discussed below.

Robust Input Submission Protocol The robust input submission protocol denoted by Π^{robust}_{Submit} must guarantee that all honest servers receive consistent secret-shares ${}^{t_s}[\![\mathsf{pkg}]\!]_q$. Such input submission protocol can be realized using offline and online phases [5, 15]. In the offline phase, recipient servers jointly computes $[\![r]\!]$ for a random value r (in the robust way, described in the next subsection) and sends the shares to a client and the client recovers r from the shares. In the online phase, the client now having secret input x to be shared computes $x-r$ and *broadcasts* it to servers. Here, a broadcast channel is a cryptographic broadcast channel where it is guaranteed for all receivers to received the same data. Then, the servers locally compute their shares $[\![x]\!] = (x-r) - [\![r]\!]$. We provide the analysis on the amount of data to be broadcasted and compare with other relevant works in Section 8. We note that Protocol 3 is a fair protocol if input message packages are submitted by Π^{robust}_{Submit} so that the input shares are guaranteed to be t_s-consistent.

Robust Double Random Protocol The robust double random functionality denoted by $\mathcal{F}^{Robust}_{DouRan}()$ takes as input a positive integer ℓ, and outputs ℓ pairs of double random sharings in a batched way regardless of an adversary's disruption strategy. In this work, we simply adopt a robust double random protocol that securely realizes $\mathcal{F}^{Robust}_{DouRan}()$ from [1, 20] which relies on an elegant non-interactive dispute control [5, 15]. Due to space restriction, we refer the reader to further discussion and protocol description in [1, 20].

For the case of small number of servers, $\mathcal{F}^{Robust}_{DouRan}()$ can realized in a non-interactive manner removing communication during the offline phase by using Pseudo-Random Secret Sharing (PRSS). In this way, servers generate double random sharings by relying only on local evaluations of pseudo-random functions on seeds pre-distributed only once at the initialization of servers [7]. We refer to Appendix F for the further discussion of corresponding parameters and efficiency.

With replacing Π_{Submit} with Π^{robust}_{Submit} and $\mathcal{F}_{DouRan}$ with $\mathcal{F}^{Robust}_{DouRan}$ in Protocol 3, we finally have our robust Rabbit-Mix protocol.

Theorem 2. *Define Π^{robust}_{AB} to be the protocol obtained from modifying Π_{AB} (Protocol 3) as follows: (1) clients submit their messages by using Π^{robust}_{submit} and (2) servers compute double random shares by running $\mathcal{F}^{Robust}_{DouRan}$. Then, for any malicious adversary corrupting up to $t_c < N_c$ clients and $t_s < N_s/4$ servers and for any client messages $m_1, m_2, \cdots, m_{N_c}$, protocol Π^{robust}_{AB} securely realizes robust anonymous broadcast functionality $\mathcal{F}_{AB}$ with the anonymity set size at least $N_c - t_c$ as defined in Definition 1.*

Proof. The security and correctness of Π^{robust}_{AB} follow from the security and correctness of Protocol 3 with the similar simulation strategy. Intuitively, the robustness is guaranteed as follows. By the robustness guarantees of $\mathcal{F}^{robust}_{DouRand}$, the preprocessed correlated randomness $({}^{t_s}[\![r_i]\!], {}^{2t_s}[\![R_i]\!])$ is guaranteed to be t_s-consistent such that all honest servers hold consistent shares. In addition, Π^{robust}_{submit} guarantees that client input shares ${}^{t_s}[\![\mathsf{pkg}^{(i)}]\!]_q$ are t_s-consistent as well. Therefore, all $[\![b^{(i)}]\!]$'s at Step 3 and all $[\![x_j]\!]$'s at Step 8 are $2t_s$-consistent regardless of an adversary's strategies so that Π^{robust}_{open} always succeeds to reconstruct all secret shares to honest servers. Therefore, Π^{robust}_{AB} always outputs honest clients' messages. □

8 Concrete Complexity Analysis

We present the in-depth communication and computation cost estimates and comparisons to relevant anonymous broadcast protocols with fairness at minimum, Blinder [1], Switch-Network and PowerMix [27].

Online Communication We present communication cost estimates: the first is the c-to-s communication cost is the number of field elements broadcast by a client to each server. Recall that the robust input submission protocol described in Section 7 requires a client to broadcast message package masked by random elements. Hence, we provide the number of field elements to be broadcast per client. The second is the per-client/per-server s-to-s communication cost which is the number of field elements to be transmitted over point-to-point authenticated (P2P) channels. See Table 2. We note that the server communication for Blinder and our protocol do not include the communication incurred by computing random coins. Random coins can be computed by either (1) opening as many random shares as required using the batch opening protocol or (2) opening a single random share and then applying a pseudo-random generator on the reconstructed randomness. We provide the required number of random coins and corresponding necessary offline resources in Table 5.

Communication vs. Blinder Note that Blinder's complexities are estimated based on the protocol with *95% message delivery success rate*. Blinder relies on the private writing (PW) paradigm similarly to Riposte [12] that employs redundancy. Blinder's server-to-server communication complexity

	t_s	Network Type	Service Strength	Delivery Success	Offline Preprocess	C-to-S (Broadcast)	S-to-S (P2P)	Server Round
Switch-Network [27]	$< \frac{N_s}{3}$	A	Fair	100%	Non-Linear (MultTri)	1	$12 N_s N_c \log^2 N_c$	$\log^2 N_c$
PowerMix [27]	$< \frac{N_s}{3}$	A	Fair	100%	Non-Linear (Powers)	1	$6{N_c}^2$	2
Blinder [1]	$< \frac{N_s}{4}$	S	Robust	95%	Linear (DouRand)	$4.97\sqrt{N_c}$	$17N_c$	4
Rabbit-Mix (This Work)	$< \frac{N_s}{4}$	S	Robust	$100-\varepsilon$%	Linear (DouRand)	$1.87\sqrt{N_c}$	$5.32N_c$	4

Table 2: Online Communication Comparisons: Let N_c and N_s be the numbers of clients and servers respectively. t_s = server corruption threshold, Network Type is either S (Synchronous) or A (Asynchronous). Protocol Feature is either Fair or Robust. The offline communication column shows the types of information that needs to be prepared during the offline phase for a protocol execution of a single epoch. DouRan, MultTri, Powers mean double random sharing, multiplication triple sharing, powers of a random sharing. See Table 5 for more in-depth offline complexity comparisons. C-to-S (Broadcast) is the number of field elements for a client to reliably broadcast to all servers. S-to-S (P2P) is the number of field elements sent *by a server to each server* via authenticated P2P channels.

	Client Computation	Server Computation
Switch-Network [27] (Fair)	C_{ss}	$(2N_cN_s\log^2 N_c)C_m + (2N_cN_s\log^2 N_c)C_a + (12\frac{N_c}{N_s}\log^2 N_c)C^M_{rec}$
PowerMix [27] (Fair)	C_{ss}	$(\frac{{N_c}^3}{2} + N_cN_s)C_m + ({N_c}^3 + \frac{{N_c}^2}{2} + N_cN_s)C_a + 6\frac{N_c}{N_s}C^M_{rec} + C_{RF}(N_c)$
Blinder [1] (Robust)	$4.97\sqrt{N_c}C_{ss}$	$(5.5{N_c}^2 + 13.2N_c\sqrt{N_c} + 5.74N_cN_s)(C_m + C_a) + 15.3\frac{N_c}{N_s}C^M_{rec} + \alpha(\log q - 2)C_m$
Rabbit-Mix (Robust)	$1.87\sqrt{N_c}C_{ss}$	$(1.43{N_c}^2 + 6.15N_c\sqrt{N_c} + 2N_cN_s)C_m + (1.5{N_c}^2 + 2N_cN_s)C_a + 5.32\frac{N_c}{N_s}C^M_{rec} + C_{RF}(N_c)$

Table 3: Online Computation Complexities: Let N_c and N_s be the numbers of clients and servers respectively. The computational complexity is measured in terms of the numbers of field operations (multiplications and additions) and of invocations of secret-sharing or share-reconstruction. C_m and C_a denote the costs of field multiplication and addition respectively. C_{ss} denotes the cost of secret-sharing. C^M_{rec} denote the costs of Reed-Solomon-decoding. $C_{RF}(N_c)$ is the cost of root-finding for N_c roots. In the above, α is the expected number of locations in the Blinder's PW table with non-zero values.

is $(6+4\gamma)N_c$ with $\gamma = 2.75$ for 95% message delivery success rate[5]. With a target message delivery success rate similar to ours (e.g., 100%), γ needs to increase accordingly (See Table 4) resulting in the increase of the communication (as well as computation) in a linear fashion in γ. Specifically, Blinder with 99.9% delivery success rate has server-to-server communication of $96N_c$ field elements per server which is 17 times bigger than Rabbit-Mix while still expectedly dropping 10 messages out of 10000 messages.

Success Rate (%)	95	96	97	98	99	99.9
Expansion Rate γ	2.75	3.14	3.67	4.63	6.71	22.02

Table 4: Redundancy and Delivery Success Rate

Online Computation We present the concrete computational cost estimates of the related work in terms of the number of field operations, multiplications and additions as well as the number of calls to the secret-share reconstruction protocol (Table 3). Our implementation of Root-finding algorithm has expected computational cost $C_{FindRoot}(N_c) \approx 7N_c\log^2 N_c$ for a fixed q. Thus, we observe that the major computational bottleneck in common is the cost to decompress clients' input shares.

Offline Requirements The performance of a system also crucially depends on the offline complexity. If the online phase requires the offline phase to prepare huge amount of preprocessed data, then the system in general cannot provide a continuous and efficient service. Table 5 provides the offline computation estimates of relevant protocols. Rabbit-Mix requires to prepare $2N_c + 0.94\sqrt{N_c}$ double random shares in the offline phase which is approximately 30% of Blinder and asymptotically improves by factor of N_c over PowerMix and Switch-Network, mixing protocols with fairness that in turns use these double random shares to prepare non-linear random shares (Multiplication Triple and Powers of random shares).

Discussion Both C-to-S and S-to-S communication of our robust AB protocol are reduced by over 60% in comparison to the latest robust AB protocol Blinder [1] while our message

[5] We assume here that the choice of prime is large enough so that a single execution of verification satisfies its info-theoretic security requirement.

	Double Random	Random Coin	Multiplication Triple	Random Bit Share	N_c-Powers of A Random Share	Total # of Double Random's
Switch-Network [27]	0	0	$1.25N_c \log^2 N_c$	$\frac{1}{2}\log^2 N_c$	0	$1.75N_c \log^2 N_c$
PowerMix [27]	0	0	0	0	$2N_c{}^2$	$2N_c{}^2$
Blinder [1]	$5.75N_c$	$N_c + 2.49\sqrt{N_c}$	0	0	0	$6.75N_c + 2.49\sqrt{N_c}$
Rabbit-Mix (This Work)	$2N_c$	$0.94\sqrt{N_c}$	0	0	0	$2N_c + 0.94\sqrt{N_c}$

Table 5: Offline Requirement Comparisons: N_c is the number of clients. The middle five columns (Double Random, Random Coin, Multiplication Triple, Random Bit Share, N_c-Powers of Random Share) show the types of preprocessed material shares required to execute each protocol per epoch. The values represents the amount of each preprocessed materials measured by the necessary numbers of Double Random Shares. The last column shows the total required number of double random shares which is the sum of all values on the left columns. Computing a preprocessed shared random coin (e.g. random shares) only requires the half of the computational cost of a double random so that the values of the column represent the half number of double random shares for the sake of measuring convenience.

delivery success rate is 100% in contrast to Blinder's 95%. The server computation cost of our protocol also improves over Blinder by over 60% in terms of number of field operations. In comparison to PowerMix, our protocol provides further optimized mixing efficiency with stronger robustness of the protocol. Essentially, our optimization allows our robust AB protocol to have $O(N_c{}^2)$ computational overhead with sublinear client computation and C-to-S communication overhead in contrast to $O(N_c{}^3)$ computational overhead of PowerMix [27]. Our AB protocol also requires minimal offline resources compared to previous relevant works as our protocol's offline resource requires less than 30% of double random shares compared to robust Blinder and PowerMix which requires the quadratic number of double randoms in the number of clients.

9 Implementation and Benchmarks

We implemented the preliminary prototype[6] of Rabbit-Mix protocol described in Section 6 and Section 7. Specifically, we implemented all online-computation subprotocols of robust Rabbit-Mix protocol (Protocol 3) with mock-up communication. That is, broadcasts and P2P channels between servers and clients are performed by direct reads and writes to each others' memory. The simulation of the Rabbit-Mix protocol is synchronized and local where servers' state machines configured with the number of servers, clients, and prime numbers execute subprotocols one by one in the state-by-state manner with mock-up communication where clients and servers directly read data from their read memory and write data to other servers' read memory. Therefore, our experimentation and benchmarks have limitations in capturing network-induced latencies and exposing bandwidth bottlenecks.

[6]This will be open-sourced at the point of publication due to an administrative reason.

Our prototype is implemented in C++ using Number Theoretic Library [35] to implement the algebraic operations. We note that, as our prototype directly uses NTL's native classes, the prototype is not to claim the best performance but to provide the insights on the protocol asymptotic behavior depending on distinct settings and parameters.[7] We tested our prototype on a virtual machine running 64-bit Ubuntu equipped with Intel Xeon CPU E3-1505M 3.00GHz with 4 CPU cores and 32 GB memory. All the benchmarks are measured based on 10 trials with 5 servers.

Computation Latency with Distinct Primes and Numbers of Clients First, we measured the server computation latency of the robust Rabbit-Mix prototype. We tested our prototype with N_c clients and distinct FFT-friendly primes of 64-1664 bits where the number of clients N_c is selected to be $N_c = 14L^2 + 10L - 1$ according to the additive 2-basis encoding to encode client messages. Specifically, we chose $N_c = 399, 1223, 2135, \cdots$ so that the number of clients are spread approximately by 800. Figure 1 presents the per-server computation latency with distinct primes.

Figure 2 shows the server computation latencies per client depending on distinct settings of number of clients, servers and primes as in the previous. As the latency measurements are scaled by the number of clients, a linear (or close-to-linear) trend can be observed as server computation is $O(N_c{}^2)$. In addition, we observe in Figure 3 that some primes yield larger throughput in comparison to other prime settings.

Indeed, Figure 3 shows that a group of primes leads to larger throughput in comparison to a 208-byte prime. Specifically, consider that clients wish to send a 208-byte message.

[7]For example, even though the Rabbit-Mix's input decompression protocol requires less than half of algebraic operations (for both multiplications and additions) that Blinder requires, our implementation has higher latency than Blinder implementation due to differences in experimentation systems and implementations. The more optimized implementation based on light-weight libraries such as GMP is a future work.

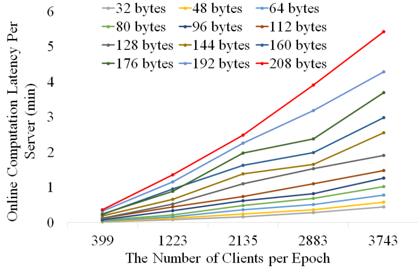

Figure 1: Per-Server Computation-Only Latency with Distinct Primes and Numbers of Clients.

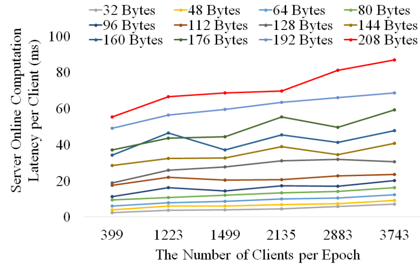

Figure 2: Per-Server Computation-Only Per-Client Latency with Distinct Primes and Numbers of Clients.

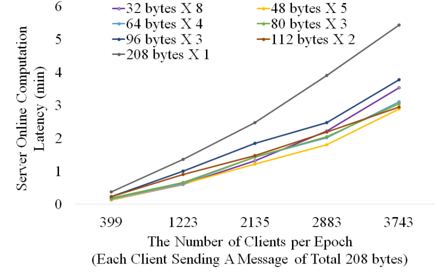

Figure 3: Per-Server Computation-Only Latency Delivering fixed-length Messages with the Distinct Prime Lengths.

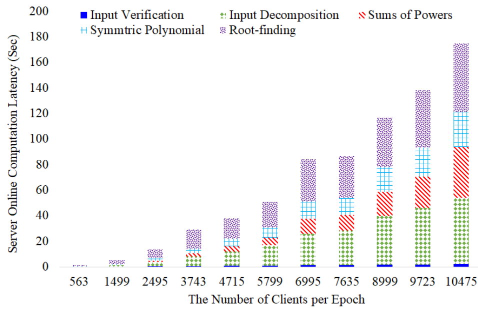

Figure 4: Server Computation Latency with Subprotocol Distribution in Logarithmic Scale: The benchmarks are measured with 5 servers executing the protocol over a 256 bit FFT-prime. Blue, orange, gray, yellow, purple colors corresponds to input format verification, input decompression, computing sum of powers, the symmetric polynomial, and the Graeffe-root finding algorithm to find roots from the symmetric protocol respectively.

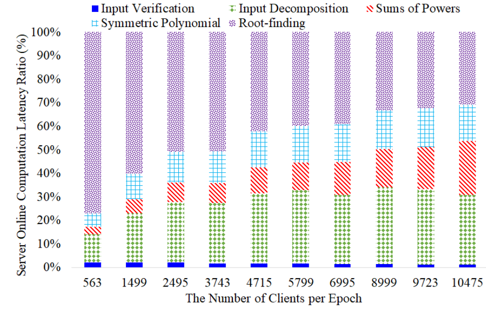

Figure 5: Server Subprotocol Computation Distribution Ratio: The ratio is based on the measurements in Figure 4.

In the first case, clients may submit a single 208-byte message to servers which run Rabbit-Mix with a 208-byte prime. In the second (and other) case(s), clients may partition its 208-byte messages into and submit μ messages of size $< 208/\mu$ to servers that in turns execute sequential Rabbit-Mix protocol μ-many times. For example, the gray line represents the total server computation latency of 8 sequential Rabbit-Mix executions with a 32-byte prime delivering a 208-byte message. Here, each message partition contains a 40-bit tag (See Section 6). Here, we set each partition to contain a 40-bit tag to reconstruct the whole messages at the end. Figure 3 essentially shows the selections of primes smaller than 208 bytes with which Rabbit-Mix delivers 208-byte messages with smaller server computation latency.

Server Computation Distribution In the following, we present the online computation latency distributions among the subprotocols of our robust ACB server. We first provide the server computation latency of Rabbit-Mix protocol depending on distinct numbers of clients spread by about 1000 up to over 10k clients per epoch with the settings of a 256-bit FFT-prime in Figure 4. Root-finding computation is observed to induce less computational time as the number of clients increases. However, it induces non-trivial computational latency due to the complexity of polynomial operations [23] for the smaller number of clients.

In Figure 5, we present the ratio between subprotocol computation latencies. We observe that the input decompression becomes the major computational bottleneck as expected when the number of clients grows. Recall that the overheads of subprotocols computing; input decompressions, sums of powers, and a symmetric polynomial is $O({N_c}^2)$, quadratic in the number of clients whereas the root-finding algorithm's overhead is $\tilde{O}(N_c)$ with a relatively large hidden constant. Therefore, when mixing larger volumes (e.g., 1 million) of client messages of relatively small sizes (e.g., 200 bytes), the computation latency from the root-finding is expected to be relatively tiny compared to the other subprotocols such as input decompression, the sums of powers, and symmetric polynomials. We note that Blinder's main computational bottleneck is also the input decompression where the Blinder protocol's input decompression requires at least 2 times more algebraic operations than ours.

Finally, Figure 6 presents concrete communication over-

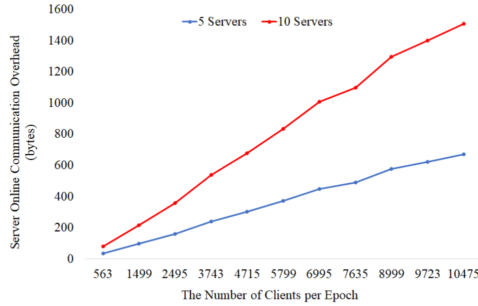

Figure 6: Server Communication Overhead in Bytes: Each server transmits the respective numbers of bytes to other servers during the online phase. The experimentation is taken with clients sending a single message of a 256-bit prime.

heads in bytes. Specifically, we measure the sizes of memory blocks (containing field elements) for each server to transmit to other servers via authenticated point-to-point channels during Rabbit-Mix's online phase. The experimentation is performed with two distinct number of servers (5 and 10 servers) with 256-bit FFT prime field.

Malicious Clients' Impact on Server Latency In the robust Rabbit-Mix protocol, clients are forced to submit consistent shares to honest servers by Π^{robust}_{submit}. Therefore, only a place that malicious clients may deviate is to submit the secret shares of a malformed message encoding that does not obey its additive-2 encoding format. In such cases, honest servers will detect those malformed encoding shares at the end of subprotocol Π_{ver} and simply drop those messages in the rest of execution with adding that client to the list of corrupted clients. Therefore, deviation by corrupted clients cannot degrade server computation latencies.

Acknowledgments

This material is based upon work supported by the United States Air Force and DARPA under Contract No. FA8750-19-C-0031. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force and DARPA. We thank the anonymous reviewers for insightful comments and suggestions.

References

[1] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder - scalable, robust anonymous committed broadcast. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1233–1252. ACM Press, November 2020.

[2] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 1217–1234. USENIX Association, August 2017.

[3] Gilad Asharov and Yehuda Lindell. A full proof of the BGW protocol for perfectly secure multiparty computation. *Journal of Cryptology*, 30(1):58–151, January 2017.

[4] Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 305–328. Springer, Heidelberg, March 2006.

[5] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 213–230. Springer, Heidelberg, March 2008.

[6] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988.

[7] Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part II*, volume 13043 of *LNCS*, pages 129–161. Springer, Heidelberg, November 2021.

[8] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[9] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

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

[11] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988.

[12] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE Computer Society Press, May 2015.

[13] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM CCS 2010*, pages 340–350. ACM Press, October 2010.

[14] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In Samuel T. King, editor, *USENIX Security 2013*, pages 147–162. USENIX Association, August 2013.

[15] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[16] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *USENIX Security 2004*, pages 303–320. USENIX Association, August 2004.

[17] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[18] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1775–1792. USENIX Association, August 2021.

[19] Michael J. Freedman and Robert Morris. Tarzan: a peer-to-peer anonymizing network layer. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 193–206. ACM Press, November 2002.

[20] Jun Furukawa and Yehuda Lindell. Two-thirds honest-majority MPC for malicious adversaries at almost the cost of semi-honest. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1557–1571. ACM Press, November 2019.

[21] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

[22] Dan Gordon. Covering designs. https://www.dmgordon.org/cover/.

[23] Bruno Grenet, Joris Van Der Hoeven, and Grégoire Lecerf. Randomized root finding over finite FFT-fields using tangent Graeffe transforms. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 197–204, 2015.

[24] Amir Houmansadr and Nikita Borisov. The need for flow fingerprints to link correlated network flows. In Emiliano De Cristofaro and Matthew K. Wright, editors, *PETS 2013*, volume 7981 of *LNCS*, pages 205–224. Springer, Heidelberg, July 2013.

[25] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul F. Syverson. Users get routed: traffic correlation on tor by realistic adversaries. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 337–348. ACM Press, November 2013.

[26] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. *SIGCOMM Comput. Commun. Rev.*, 43(4):303–314, aug 2013.

[27] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew K. Miller. HoneyBadgerMPC and AsynchroMix: Practical asynchronous MPC and its application to anonymous communication. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 887–903. ACM Press, November 2019.

[28] D. G. Mead. Newton's identities. *The American Mathematical Monthly*, 99(8):749–751, 1992.

[29] D. G. Mead. Newton's identities. *The American Mathematical Monthly*, 99(8):749–751, 1992.

[30] Prateek Mittal and Nikita Borisov. ShadowWalker: peer-to-peer anonymous communication using redundant structured topologies. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 2009*, pages 161–172. ACM Press, November 2009.

[31] Svein Mossige. Algorithms for computing the h-range of the postage stamp problem. *Mathematics of Computation*, 36(154):575–582, 1981. Appendix by Torleiv Kløve and Svein Mossige.

[32] Oleg Pikhurko. Dense edge-magic graphs and thin additive bases. *Discrete mathematics*, 306(17):2097–2107, 2006.

[33] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2P mixing and unlinkable bitcoin transactions. In *NDSS 2017*. The Internet Society, February / March 2017.

[34] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[35] Victor Shoup. Ntl: A library for doing number theory. http://www.shoup.net/ntl/.

[36] Alexandre Soro and Jerome Lacan. Fnt-based reed-solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5, 2010.

[37] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. RAPTOR: Routing attacks on privacy in tor. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 271–286. USENIX Association, August 2015.

[38] Adithya Vadapalli, Kyle Storrier, and Ryan Henry. Sabre: Sender-anonymous messaging with fast audits. In *2022 IEEE Symposium on Security and Privacy*, pages 1953–1970. IEEE Computer Society Press, May 2022.

[39] Arnulf Von Mrose. Untere schranken für die reichweiten von extremalbasen fester ordnung. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 48-1, pages 118–124. Springer, 1979.

[40] 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, Hollywood, CA, October 2012. USENIX Association.

[41] Doron Zeilberger. A combinatorial proof of newton's identities. *Discret. Math.*, 49:319, 1984.

A Additive 2-basis Decompression Circuit

Let $N = 14L^2 + 10L - 1$ for a positive integer L and $\mathcal{B}$ be the additive 2-basis for $[N]$. Then the following circuit shows how to decompress $\mathcal{B}$ into $[N]$.

Circuit 1: Additive 2-basis Decompression Circuit $\mathsf{Decomp}(\beta)$

Input: Let $N = 14L^2 + 10L - 1$. The input is $\mathcal{B} = \{S_1, S_2, S_3, S_4, S_5\}$ as above.
Output: $[N] = \{1, 2, \cdots, N\}$.

1: **Elements from** 1 **to** $2L$**:** We already have elements from 1 to L. We can populate the rest by computing $i + L$ for each $i \in L$. This requires L additions in total.
2: **Elements from** $2L+1$ **to** $3L^2+2L$**:** These elements can be computed as $i + j$ for each $i \in S_1$ and $j \in S_2$. The total number of additions is $3L^2$.
3: **Elements from** $3L^2+2L+1$ **to** $4L^2+3L-1$**:** The elements in this range can be computed also by computing additions between S_1 and S_3 as $i + j$ for each $i \in S_1$ and $j \in S_3$. The total number of additions is L^2.
4: **Elements from** $4L^2+3L$ **to** $6L^2+4L-1$**:** These elements can be computed by computing additions of elements of $i \in S_3$ and $j \in S_2' \subset S_2$ where the subset $S_2' = \{L^2 + Li' - Lj' \mid i' \in [2L+1] \text{ and } j' \in \{0, 1, \cdots, L-1\}\}$. Let $s_{3,j}$ be the j-th number of S_3. Then, the circuit populates the elements in the range by adding for each $i \in [2L+1]$ and each $j \in [1, \cdots, L]$, $s_{3,j} + L^2 + Li - L(j-1)$. The total number of additions is $2L^2 + L$.
5: **Elements from** $6L^2+4L$ **to** $6L^2+5L$**:** S_4 trivially covers this range.
6: **Elements from** $6L^2+5L+1$ **to** $6L^2+6L$**:** The elements in the range can be computed by additions between S_1 and $6L^2+5L$, the last element of S_4, thus with L additions.
7: **Elements from** $6L^2+6L+1$ **to** $9L^2+6L$**:** The elements in this range can be computed by additions between S_2 and S_4 where the number of additions is $3L^2$.
8: **Elements from** $9L^2+6L+1$ **to** $10L^2+7L-1$**:** This interval can be computed by additions between S_3 and S_4 where the number of additions is L^2+L.
9: **Elements from** $10L^2+7L$ **to** $10L^2+9L$**:** The elements up to $10L^2+8L$ in this interval is already covered by S_5 and the rest can be computed by additions between S_1 and $10L^2+9L$. Hence, the number of additions is L.
10: **Elements from** $10L^2+9L+1$ **to** $13L^2+9L$**:** The interval is computed by adding elements between S_2 and S_5. Hence, the number of additions is $3L(L+1)$.
11: **Elements from** $13L^2+9L+1$ **to** $14L^2+10L-1$**:** Finally, the interval can be computed by adding elements between S_3 and S_5 and the number of additions is L^2+L.

B Client-Input Encoding Decompression

The following protocol shows how to decompress the secret-shares of client message package pkg into the secret-shares of N_c powers of a message. In the following, we have $N_c = 14L^2 + 10L - 1$ for some positive integer L and denote a t-consistent secret-shared encoding of client message m by ${}^t[\![\mathsf{pkg}]\!]_q$ where $\mathsf{pkg} = m^{\mathcal{B}}$.

Protocol 4: Client Input Decompression Protocol $\Pi_{Decomp}({}^t[\![\mathsf{pkg}^{(i)}]\!]_q)$

Parameters: N_c and L such that $N_c = 14L^2 + 10L - 1$. The number of servers is N_s. t is the threshold on the number of corrupted servers. q is a prime number.
Preprocessed Information: None
Input: A mixing server S_j has as inputs the j-th consistent t-sharings of input message encoding ${}^t[\![\mathsf{pkg}^{(i)}]\!]_q \in \mathbb{F}_q{}^{7L+2}$ received from client C_i with input message m_i such that ${}^t[\![\mathsf{pkg}^{(i)}]\!]_q = \left\{{}^t[\![S_1^{(i)}]\!]_q, {}^t[\![S_2^{(i)}]\!]_q, \cdots, {}^t[\![S_5^{(i)}]\!]_q\right\}$, where $\mathsf{pkg}^{(i)}$ is submitted via Protocol 2.

Output: An ordered set of N_c secret-shares $[\![x^{(i)}]\!]_q$ defined as

$$\{{}^{t}[\![x_1^{(i)}]\!]_q, \cdots, {}^{t}[\![x_L^{(i)}]\!]_q, \\ {}^{2t}[\![x_{L+1}^{(i)}]\!]_q, \cdots, {}^{2t}[\![x_{6L^2+4L-1}^{(i)}]\!]_q, \\ {}^{t}[\![x_{6L^2+4L}^{(i)}]\!]_q, \cdots, {}^{t}[\![x_{6L^2+5L}^{(i)}]\!]_q, \\ {}^{2t}[\![x_{6L^2+5L+1}^{(i)}]\!]_q, \cdots, {}^{2t}[\![x_{10L^2+7L-1}^{(i)}]\!]_q, \\ {}^{t}[\![x_{10L^2+7L}^{(i)}]\!]_q, \cdots, {}^{t}[\![x_{10L^2+8L}^{(i)}]\!]_q, \\ {}^{2t}[\![x_{10L^2+8L+1}^{(i)}]\!]_q, \cdots, {}^{2t}[\![x_{N_c}^{(i)}]\!]_q\}.$$

Protocol: Each server locally computes the following:

1: **Every j from 1 to $2L$:** We already have output shares from 1 to L so that set $[\![x_j^{(i)}]\!] = [\![s_{1,j}^{(i)}]\!]$ where $[\![s_{1,j}^{(i)}]\!]$ is the j-th share in the input share set $[\![S_1^{(i)}]\!]$. Compute the rest ($j \in [L+1, \cdots, 2L]$) by computing ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{1,j-L}^{(i)}]\!] \cdot {}^{t}[\![s_{1,L}^{(i)}]\!]$.

2: **Every j from $2L+1$ to $3L^2+2L$:** Compute for each for $l \in [3L]$ and $k \in [L]$, ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{1,k}^{(i)}]\!] \cdot {}^{t}[\![s_{2,l}^{(i)}]\!]$.

3: **Every j from $3L^2+2L+1$ to $4L^2+3L-1$:** For each $k \in [L]$ and $l \in [L]$, compute ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{1,k}^{(i)}]\!] \cdot {}^{t}[\![s_{3,l}^{(i)}]\!]$.

4: **Every j from $4L^2+3L$ to $6L^2+4L-1$:** For each $k \in [2L+1]$ and each $l \in [L]$, compute ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{2,L+k-l}^{(i)}]\!] \cdot {}^{t}[\![s_{3,l}^{(i)}]\!]$.

5: **Every j from $6L^2+4L$ to $6L^2+5L$:** Set for each $k \in [L+1]$, ${}^{t}[\![x_j^{(i)}]\!] = {}^{t}[\![S_{4,k}^{(i)}]\!]$.

6: **Every j from $6L^2+5L+1$ to $6L^2+6L$:** For each $k \in [L]$, compute ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![S_{1,k}^{(i)}]\!] \cdot {}^{t}[\![S_{4,L+1}^{(i)}]\!]$.

7: **Every j from $6L^2+6L+1$ to $9L^2+6L$:** For each $k \in [3L]$ and each $l \in [2, L+1]$, compute ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![S_{2,k}^{(i)}]\!] \cdot {}^{t}[\![S_{4,l}^{(i)}]\!]$.

8: **Every j from $9L^2+6L+1$ to $10L^2+7L-1$:** For each $k \in [L]$ and each $l \in [L+1]$, skip the first computation ${}^{t}[\![S_{3,1}^{(i)}]\!] \cdot {}^{t}[\![S_{4,1}^{(i)}]\!]$ and then compute the rest as ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![S_{3,k}^{(i)}]\!] \cdot {}^{t}[\![S_{4,l}^{(i)}]\!]$.

9: **Every j from $10L^2+7L$ to $10L^2+9L$:** For $k \in [L+1]$, set ${}^{t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{5,k}^{(i)}]\!]$. Then, compute for $k \in [L]$, compute ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{1,k}^{(i)}]\!] \cdot {}^{t}[\![s_{5,L+1}^{(i)}]\!]$.

10: **Every j from $10L^2+9L+1$ to $13L^2+9L$:** For each $k \in [3L]$ and $l \in [2, L+1]$, compute ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{2,k}^{(i)}]\!] \cdot {}^{t}[\![s_{5,l}^{(i)}]\!]$.

11: **Every j from $13L^2+9L+1$ to $14L^2+10L-1$:** For each $k \in [L]$ and $l \in [L+1]$, skip the first computation ${}^{t}[\![s_{3,1}^{(i)}]\!] \cdot {}^{t}[\![s_{5,1}^{(i)}]\!]$ and then compute the rest ${}^{2t}[\![x_j^{(i)}]\!] = {}^{t}[\![s_{3,k}^{(i)}]\!] \cdot {}^{t}[\![s_{5,l}^{(i)}]\!]$.

C Input Well-formedness Verification Circuit

The following circuit describes the algebraic circuit to compute a share of linear sketch predicate indicating whether a client's input message encoding is well-formed according to additive 2-basis for the number of clients N_c.

Circuit 2: Well-formedness Verification Circuit $\mathsf{FormatVerify}()$

Input Parameters: Let q be a prime. Let $N_c = 14L^2 + 10L - 1$ be the number of participating clients in the current epoch.
Input: A set of $7L+4$ field elements $X = \{x_1, \cdots, x_{7L+4}\}$ to be verified which can be partitioned into 6 ordered sets X_1, X_2, X_3, X_4, X_5 and X_6. A set of random field elements $\{c_i\}_{i\in[7L+3]}$ sampled independent of X.
Output: Output b where $b = 0$ if and only if pkg satisfies both the inner progression and inter progress conditions as defined above.

1: Parse the input set X into 6 ordered subsets $X_1, X_2, \cdots, X_6$ where $X_1, X_2, \cdots, X_5$ are ordered sets of elements having additive 2-basis S_1, S_2, $\cdots$, S_5 as exponents accordingly. We let X_6 contain the final two elements of X.
2: Compute $b_1^0 = \sum_{i=1}^{L-1} c_i(x_{1,i+1} - x_{1,i}x_{1,1})$
3: Compute $b_2^0 = \sum_{i=1}^{3L-1} c_{i+L-1}(x_{2,i+1} - x_{2,i}x_{1,L})$
4: $b_3^0 = \sum_{i=1}^{L-1} c_{i+4L-2}(x_{3,i+1} - x_{3,i}x_{6,1}) + c_{5L-2}(x_{6,1} - x_{1,1}x_{1,L})$
5: $b_4^0 = \sum_{i=1}^{L} c_{i+5L-2}(x_{4,i+1} - x_{4,i}x_{1,1})$
6: $b_5^0 = \sum_{i=1}^{L} c_{i+6L-2}(x_{5,i+1} - x_{5,i}x_{1,1})$
7: $b_1^1 = 0$
8: $b_2^1 = c_{7L-1}(x_{1,L}x_{1,L} - x_{2,1})$
9: $b_3^1 = c_{7L}(x_{2,3L}x_{1,L} - x_{3,1})$
10: $b_4^1 = c_{7L+1}(x_{4,1} - x_{3,1}x_{3,1})$
11: $b_5^1 = c_{7L+2}(x_{5,1} - x_{6,2}x_{1,1}) + c_{7L+3}(x_{6,2} - x_{3,L}x_{4,L+1})$
12: Compute and output $b = \sum_{i=0}^{1}\sum_{j=1}^{5} b_j^i$.

D Well-formedness Verifiable Client Input Submission

In order for (shared) client input encodings to be verifiable, an additive 2-basis $\mathcal{B}$ now modified to contain additional set of elements $S_6 = \{s_{6,1}, s_{6,2}\} = \{L+1, 10L^2+7L-1\}$ as described in Section 5. Hence, the new input submission protocol is identical a client with message m and the new $\mathcal{B} = \{S_1, S_2, \cdots, S_6\}$ creates $m^{\mathcal{B}}$ and secret-shares them towards servers. We will simply use the identical notation Π_{submit} for this input submission protocol.

Protocol 5: Well-formedness Verifiable Input Submission Protocol Π_{submit}

Parameters: Prime q, the number of clients N_c, the number of servers N_s, the server corruption threshold t_s, and the additive 2-basis $\mathcal{B} = \{S_1, S_2, \cdots, S_6\}$.
Input: Client C holds a message m defined over $\mathbb{F}_q$.
Output: Servers output the secret-shares of ${}^{t_s}[\![\mathsf{pkg}]\!]_q$.

1: C computes $\mathsf{pkg}_m \leftarrow m^{\mathcal{B}}$.
2: C computes ${}^{t_s}[\![\mathsf{pkg}]\!]_q \leftarrow \mathsf{Share}_{N_s,t_s,q}(\mathsf{pkg}_m)$.
3: C sends the i-share of ${}^{t_s}[\![\mathsf{pkg}]\!]_q$ to S_i.

E Input Well-formedness Verification Protocol

The following protocol describes the protocol that takes as inputs the t-consistent sharings of a client's input message encoding and outputs a $2t$-consistent sharing of a linear sketch that indicates the well-formedness of the input message encoding. The output sharing is re-randomized by using a random-double sharing. The protocol uses Circuit 2 as a sub-module.

Protocol 6: Well-formedness Verification Protocol $\Pi_{ver}()$

Parameters: Let q be a prime. Let $N_c = 14L^2 + 10L - 1$ be the number of participating clients in the current epoch.
Preprocessed Information: A pair of *consistent* random double sharings $({}^{t}[\![r]\!], {}^{2t}[\![R]\!])$.
Input: For each $i \in [N_c]$, Client C_i's degree-t sharing of input package ${}^{t}[\![\mathsf{pkg}^{(i)}]\!]_q = \left\{[\![x_1^{(i)}]\!], \cdots, [\![x_{7L+5}^{(i)}]\!]\right\}$. Finally, a set of random challenge $\{c_i\}_{i\in[7L+4]}$ over $\mathbb{F}_q$.
Output: Output ${}^{2t}[\![b^{(i)}]\!]_q$.

1: Each server locally computes a linear sketch by evaluating $\mathsf{FormatVerify}([\![\mathsf{pkg}^{(i)}]\!], \{c_i\}) \rightarrow {}^{2t}[\![b^{(i)}]\!]_q$.
2: All servers rerandomize ${}^{2t}[\![b^{(i)}]\!]_q$ using the preprocessed random double sharing.

F Removing Offline Communication

As an alternative approach for generating robust random shares, we can make use of the recent improvements in pseudorandom secret sharing (PRSS) due to Benhamouda, Boyle, Gilboa, Halevi, Ishai, and Nof [7]. Using PRSS, participants can generate robust shares and robust double random shares locally after an initial setup step for distributing PRF seeds. This approach is worse asymptotically as N_s and the associated corruption threshold and polynomial degree grow, but it is likely to be more efficient in cases where a bounded number of servers N_s wish to perform a series of mixes for a large group of clients. For a fixed number of servers, using PRSS attains the Server-to-Server communication and round complexity in exchange for a $O(1)$ setup cost and a $O(N_c)$ computational cost.

More precisely, the online communication cost using the PRSS approach is always zero, while the associated setup time, memory costs, and online computational costs depend linearly on the minimum known size of a combinatorial object known as a (n,m,t) cover. When $t = \Theta(n)$, this minimum size is exponential in n, and so the approach is not feasible in the asymptotic case.

Nevertheless, for many concrete values of n, the resulting parameters are efficient, as we show in Table 6. For example, generating the $2N_c$ double shares required for our robust server requires around $81N_c$ PRF calls per server with 8 servers and $564N_c$ PRF calls per server with 12 servers, which are both likely to give a small increase in computation compared to the expression given in Table 3 for, say, $N_C = 100$. In many concrete settings, this increase in computation will be more than offset by the $2.7x$ improvement in communication from removing the interaction required to generate the robust shares.

N_s	d	t_s	PRSS seeds	Seed storage	PRF calls per double share
7	2	1	12.9	64.3	32.1
8	2	1	13.5	81.0	40.5
11	3	2	54.3	411.9	206.0
12	3	2	66.0	564.0	282.0
15	4	3	264.7	2785.3	1392.7
16	4	3	348.8	3975.0	1987.5
19	5	4	1686.9	22866.9	11433.5
20	5	4	2059.2	29867.4	14933.7
48	15	4	2746.4	86747.2	7228.9
72	23	4	5717.2	271813.5	13590.7

Table 6: The online and offline costs of generating packed double secret shares over polynomials of degree d and $2d$ for a given number of servers N_s and a corruption threshold t_s. All costs are given on a per-party basis. The offline costs are measured in PRSS seeds per party, which measures offline setup time, and PRSS seed storage per party, which measures memory cost. PRF calls per double share measures online computational time per party.

We require $2d + 2t_s < N_s$, and in all rows but the last two, choose t_s maximal subject to the condition $t_s < N_s/4$. The numbers in this table can be derived from Theorems 3.3 and 3.6 in [7] along with the list of covering designs maintained by Dan Gordon [22]. We include the last two rows of the table as a demonstration of how PRSS can continue to be efficient for much larger values of N_s when we set an even lower corruption threshold.